Sistemas reativos

Não confundir com sistemas radioativos

Casa do Código | alura

GUILHERME VIRGS MORAES

Sumário

- ISBN
- Prefácio por Alexandre Aquiles
- Prefácio por Biharck Araújo
- Agradecimentos
- Sobre o autor
- Sobre o livro
- Parte 1: A qualidade e o propósito
  - 1 O bom e o ruim
- Parte 2: Mais uma vez microsserviços...
  - 2 Afinal, o que são microsserviços?
  - 3 Características dos microsserviços
  - 4 Microsserviços e a Orientação a Objetos
- Parte 3: Sistemas reativos
  - 5 Um discurso de ódio à proatividade
  - 6 A Reatividade diz "olar"
  - 7 Dificuldades da reatividade
  - 8 Orientação a Objeto Orientada a Mensagens Reativas
- Parte 4: Orientação a Mensagens
  - 9 Mensagens distribuídas
  - 10 Controle de vazão
  - 11 Mensagens poliglotas
  - 12 Por que (de novo) REST/http?
- Parte 5: Elasticidade
  - 13 Distribuição elástica
  - 14 Jogo de dados
  - 15 Inconsistência distribuída
  - 16 O inferno da consistência
- Parte 6: Resiliência
  - 17 Fluxos resilientes
  - 18 Transações distribuídas
  - 19 Coordenação de fluxos
  - 20 Observabilidade de sistemas

- Parte 7: Responsividade
  - 21 O negócio e a responsividade
- Parte 8: Considerações finais
  - 22 Fim
  - 23 Bibliografia

ISBN

Impresso: 978-65-86110-82-1

Digital: 978-65-86110-83-8

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Prefácio por Alexandre Aquiles

Este é um livro peculiar.

Não apenas por abordar conceitos profundos de assuntos complexos como Arquitetura de Software e Sistemas Distribuídos. Também não somente pela abundância de imagens (e de citações). Ou pelas linhas de código que, apesar de poucas, vão alterar sua percepção sobre programação. Nem por definir uma lei e fundar um novo paradigma!

A principal singularidade deste livro é o humor. Já leu um livro técnico em que você gargalhou? Não estou falando de uma risadinha não, mas de rachar de rir mesmo! Qual livro técnico cita Simone, Shrek, Matrix, Xuxa, Chaves, um ex-BBB e até a famigerada final do Paulistão de 1999?

E, meio sem perceber, você ampliará sua perspectiva sobre software e conhecerá o Quarteto Fantástico da Reatividade!

A pergunta que fica é: já chegamos lá?

Alexandre Aquiles, Tech Lead no Grupo Alura

Prefácio por Biharck Araújo

Para algumas pessoas, leituras técnicas são como um momento de conforto e relaxamento. Mesmo que a leitura entre em detalhes sobre sistemas reativos, ela pode ser ainda mais prazerosa quando conectada a uma linguagem leve, corriqueira e divertida.

Guilherme Moraes (Virgs) desenvolveu todo um storytelling de modo que os leitores e leitoras se sentem envolvidos durante todo o contexto do livro. Em momentos em que o conteúdo se torna denso, uma piada esperta traz de volta a leveza ao conteúdo, que é riquíssimo e muito limitado principalmente em nosso idioma.

O conteúdo é bem completo, passando por introduções de microsserviços, contextos, dificuldades de sistemas reativos e desempenho até se desdobrar no mundo da elasticidade, resiliência e responsividade *per se*.

Para os que tiveram o prazer em trabalhar com Virgs, este livro é uma viagem, é como se estivessem trabalhando ou tomando um café com ele. E, para aqueles que ainda não o conhecem, esta é uma oportunidade de sentir como seria o dia a dia trabalhando ao seu lado. Que, por sinal, é exatamente assim.

Quem sabe em um futuro próximo teremos algo sobre Rubik!

Biharck Araújo, Engenheiro de Software e autor do livro Hands-On RESTful Web Services with TypeScript 3.

Agradecimentos

Como não poderia ser diferente, primeiramente gostaria de agradecer à família Balbino. Em especial, ao meu amigo Miguel Balbino, por ser uma pessoa cabriocárica e estrogonoficamente sensível.

Agradeço também aos meus familiares e amigos pelos valiosíssimos retornos e opiniões sobre o conteúdo do livro. Todos eles foram sumariamente ignorados.

Às empresas e aos times com os quais tenho trilhado minha jornada profissional que, com a criatividade de um gênio renascentista, propuseram problemas cada vez mais cabeludos e debates escaldantes. Matérias-primas indispensáveis para muitas soluções apresentadas aqui.

À Casa do Código e sua equipe por acreditar na proposta e aceitar correr o risco apostando em mim.

À minha amada esposa, musa inspiradora e fonte inesgotável de compreensão, por exercer papel indispensável no nosso casamento e ter me poupado de muitas obrigações domésticas durante a escrita do livro.

Por último, mas não menos importante, agradeço aos italianos pela pizza.

Sobre o autor

Nascido e criado em Fortaleza (CE), ex-animador de batismo e eterno meia-atacante nas peladas no Juraci às sextas-feiras à noite, comecei a tentar desenvolver sistemas com apenas 13 anos e hoje, orgulhosamente, digo que ainda estou aprendendo.

Com mais da metade da minha vida imersa no mundo do desenvolvimento de software, eu me formei em Ciências da Computação com ênfase em Engenharia de Software, sou especialista em Engenharia de Software Ágil e 32% mestre em Computação Aplicada que, pelo andar da carruagem, só devo concluir por volta de 2045.

Ao longo desse período, acumulei experiência, amizades e, principalmente, dívidas financeiras Brasil afora. Tive também a oportunidade de trabalhar em empresas das quais me orgulho muito, como ThoughtWorks, iFood e, atualmente, Amazon Web Services (AWS) - Canadá. Dedico muito do meu tempo ao desenvolvimento de uma infinidade de projetos pessoais (saiba mais em: https://pagehub.me/virgs e https://github.com/virgs). Acredito que essas honrarias e muitas aulas e palestras ministradas me credenciam para compartilhar conhecimento e alegria nas páginas que virão a seguir.

Além disso, sou profundamente viciado em cubos mágicos e em jogos não *mainstream*, como campo minado, paciência, sudoku e futebol. Mas prefiro não falar sobre o último por ainda carregar comigo a mágoa de nunca ter sido convocado para a seleção brasileira.

Sobre o livro

Com cada vez mais conexões, mais dados e usuários mais ranzinzas – agradeça à geração dos millennials –, manter a responsividade de um sistema de microsserviços se tornou uma tarefa dolorosa, exatamente no meio do caminho entre a depressão e o desespero. Por sorte, isso não precisa ser assim. É para amenizar essa dor que existem os **sistemas reativos**.

Por meio da elasticidade e resiliência, os sistemas reativos promovem a responsividade e tentam satisfazer os anseios de usuários cada vez mais exigentes - o que deve ser o objetivo de qualquer empresa que deseja competir no mercado atual.

De maneira não usual e com doses cavalares de um humor duvidoso, este livro apresenta conceitos como consistência, disponibilidade e observabilidade para amantes da computação, especialmente para aqueles que são, como eu, apaixonados pela arquitetura de microsserviços. O conteúdo apresentado, a escassez de linhas de código e parágrafos práticos fazem com que o livro seja mais bem apreciado por quem já possui um conhecimento mínimo prévio sobre arquitetura de microsserviços, o que o torna não recomendado para completos iniciantes.

Aquilo que você provavelmente já sabe e aplica no seu dia a dia passará a ser aplicado também nos sistemas que desenvolve: a cômoda **perspectiva reativa**. A aplicação dessa perspectiva garantirá que, após a leitura, seus sistemas sejam mais responsivos e, principalmente, que você nunca mais reaja da mesma maneira ao escutar a seguinte pergunta: "A gente já chegou?".

Parte 1: A qualidade e o propósito

Capítulo 1
O bom e o ruim

Figura 1.1: Ecce homo por Cecilia Giménez.

No ano de 2012, a pacata cidade Borja, na Espanha, e o mundo das artes foram violentamente chacoalhados com uma inovadora restauração.

O que aconteceu foi que, como muitos de vocês sabem, uma singela idosa resolveu restaurar a antiga obra religiosa *Ecce Homo* por conta própria, "sem pedir permissão", mas "com boas intenções". O detalhe é que restauração, digamos, não era o ponto forte do seu talento e ela tinha consciência disso.

Ela contava com sua fé inabalável e esperava que isso fosse suficiente para que uma incorporação divina tomasse seu corpo e a inspirasse na restauração, *à la* Patrick Swayze e Whoopi Goldberg no inesgotável clássico *Ghost — do outro lado da vida*.

A restauração acabou saindo pela culatra e o resultado ficou diferente da obra original e completamente irreconhecível. Entretanto, o tiro pela culatra também saiu pela culatra e produziu um notório feito de *marketing* — quem sabe a idosa tenha sido incorporada por um artista publicitário?

O fato ganhou proporções mundiais e catapultou a então desconhecida cidade ao mundo do estrelato, recebendo dezenas de milhares de turistas sedentos para testemunhar a obra. Administrada pelo município, a instituição passou a cobrar pela entrada para financiar a conservação da pintura e suas obras de caridade.

Para finalizar, a cereja desse bolo feito todo de cerejas, a idosa, receptáculo de espíritos, ao perceber no que tinha resultado sua aventura, alegou direitos autorais e recebe praticamente metade de tudo o que é arrecadado por conta da obra.

Cá entre nós, dá para dizer que o feito não foi bom?

1.1 O propósito do código

Quando falamos de qualidade de códigos de programação, imediatamente nos vêm à cabeça termos como **coesão, acoplamento, legibilidade, padrões de projeto** e uma infinidade de *buzzwords*.

Com a liberdade que só meu próprio livro me proporciona, questiono: quem disse que o código bom é aquele que é de fácil compreensão e manutenção?

Antes mesmo que eu possa finalizar meu questionamento, sei que serei interrompido com um: "Uncle Bob, no livro *Código limpo*", e a famosíssima imagem:

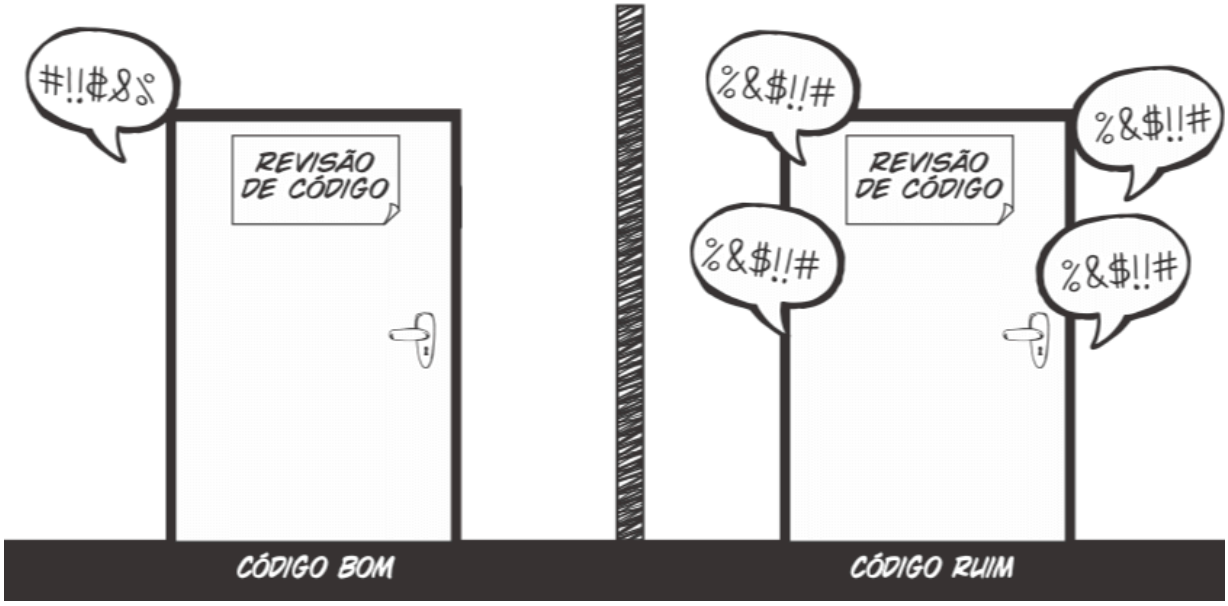

Figura 1.2: Qualidade do código medida em xingamentos por minuto.

Não ouso discordar disso. Apesar de, normalmente, ser o que se procura quando um código é escrito, complemento dizendo que existem também outros objetivos e os termos anteriores não são os principais adjetivos procurados na avaliação de sua qualidade.

Sim, existem códigos cujo único propósito é repousar eternamente no fundo do lixo. Ponto. Muitos outros, contudo, possuem propósitos diferentes. Contrariando a convenção social, é um engano associar

imediatamente a qualidade à legibilidade, à quantidade de palavrões por minuto de leitura e à facilidade de manutenção.

Para reforçar o meu argumento, apresento o seguinte código:

```
// Código de qualidade questionável
function multiplication_func(numerador, d) {
return    numerador /d
}
```

Ele é muito bom para o seu propósito: servir de exemplo.

Podemos questionar a sua implementação, limpeza, praticidade, efetividade, segurança, padronização e tal. Mas temos que aceitar que ele é um excelente código de exemplo. Portanto, serve a seu propósito e reforça meu ponto.

Sob essa ótica, percebemos que, para analisarmos a qualidade do código, temos que entender o seu propósito.

> A qualidade do código está estritamente relacionada ao propósito do código.

1.2 O código bom

Robert Nystrom, no seu recomendadíssimo livro de estreia *Game Programming Patterns*, nos dá uma ducha de sabedoria e elenca três objetivos na escrita de códigos (https://gameprogrammingpatterns.com/). Todos giram em torno da otimização da velocidade. São eles **velocidade de escrita**, **velocidade de execução** e **velocidade de compreensão.**

Essas finalidades são relacionadas de tal modo que priorizar uma pode acarretar no detrimento de outra, fazendo com que raramente as atinjamos simultaneamente. Ainda assim, o código não deixa de

ser melhor ou pior se buscarmos uma finalidade e sacrificarmos outra.

Velocidade de escrita

Existe sempre a pressão de terminar a tarefa o mais rápido possível deixando todo o resto para depois. Se nós socarmos tantas funcionalidades quanto possível em nosso projeto, ele se tornará uma pilha de confusão, gambiarras, bugs e inconsistências que afetarão a produtividade no futuro.

Mas há valor no código escrito rapidamente. É comum escrever código sabendo que ele será jogado fora. Se o propósito é testar se a ideia funciona, projetá-la com maestria significa torrar mais dinheiro antes de colher algum resultado. Se a ideia não se mostrar promissora, o tempo perdido esculpindo um código elegante é desperdiçado quando ele é jogado fora.

Amontoar código que é simplesmente funcional o suficiente para responder perguntas-chaves é uma prática perfeitamente legal e tem nome e sobrenome: **Prova de Conceito**, mas também atende pela alcunha de: **prototipação**.

Em determinados momentos, é até melhor que o resultado tenha baixíssima fidelidade ao que se espera do resultado, assim é mais difícil cair na tentação de reusar o que nasceu como prototipação para "acelerar" a entrega.

Kent Beck, lendário pioneiro da engenharia de software, nos ilumina com sua lanterna da erudição e identifica a inevitável fase da **exploração** como a primeira das três fases de maturidade de um produto, sendo as fases seguintes: **expansão** e **extração**.

Segundo ele, é na fase da exploração que começa a busca arriscada por um retorno viável que não seja o aprendizado ou a validação de uma hipótese de produto. Enquanto não temos a validação, a estratégia que agrega maior valor ao projeto é reduzir o custo de experimentação. Ainda segundo Beck, essas fases são sequenciais,

portanto otimizar outro aspecto que não seja a velocidade de escrita seria uma ação prematura prejudicial para a descoberta de um produto viável.

Além da prototipação, existem outras circunstâncias onde a velocidade da escrita é uma das principais diretrizes por trás do objetivo. Por exemplo, competições de programação ou palestras com criação de código-fonte ao vivo, com o tempo contado, quando a intenção do palestrante tende a ser exemplificar como o conceito funciona e não a preocupação com a beleza do código.

Já presenciei, inclusive, vários códigos que sequer compilavam sendo apresentados em conferências e eu, como apresentador, não exibi o mínimo de constrangimento por isso.

Velocidade de execução

A implementação mais rápida de ser escrita raramente é a de melhor desempenho. Otimizar performance requer tempo de experimentação e exploração para estudá-la.

Infelizmente, o que funciona bem em uma aplicação pode não funcionar em outra. O ideal é possuir um leque de alternativas e testar as combinações entre si até encontrar a melhor.

Por exemplo, pode ser que a substituição da estrutura de dados seja uma ótima opção. Às vezes, minimizar acessos ao disco pode ser um enorme alívio no desempenho. Há momentos em que a própria configuração de hardware pode limitar a beleza do código.

Onde eu costumava trabalhar, em uma empresa cuja especialidade eram os softwares embarcados, essa prática era muito recorrente e a chamávamos, carinhosamente, de **escovação de *bits***.

Provavelmente porque devíamos polir cada minúsculo pedaço do código a fim de torná-lo mais veloz. Eram cenários em que cada milésimo de segundo contava e muitos palavrões por minuto eram vociferados enquanto o código era lido.

Esse segundo objetivo contrapõe o primeiro e também o terceiro. Um código de desempenho extremo tende a dificultar uma eventual mudança posterior. Sempre que for necessário fazer uma adição ao código, mais tempo será despendido para escrevê-lo.

Maximizar a velocidade de execução também pode ocasionar um prejuízo na velocidade de compreensão do código. Abrir mão do polimorfismo e elegância em prol do desempenho parece um preço razoável a se pagar para economizar ciclos computacionais.

Às vezes, a depender do objetivo, o código pode se tornar um verdadeiro teste de QI, e a carga cognitiva para sua compreensão pode ser altíssima. Ainda assim, o código pode ser perfeitamente bom porque atende ao seu propósito.

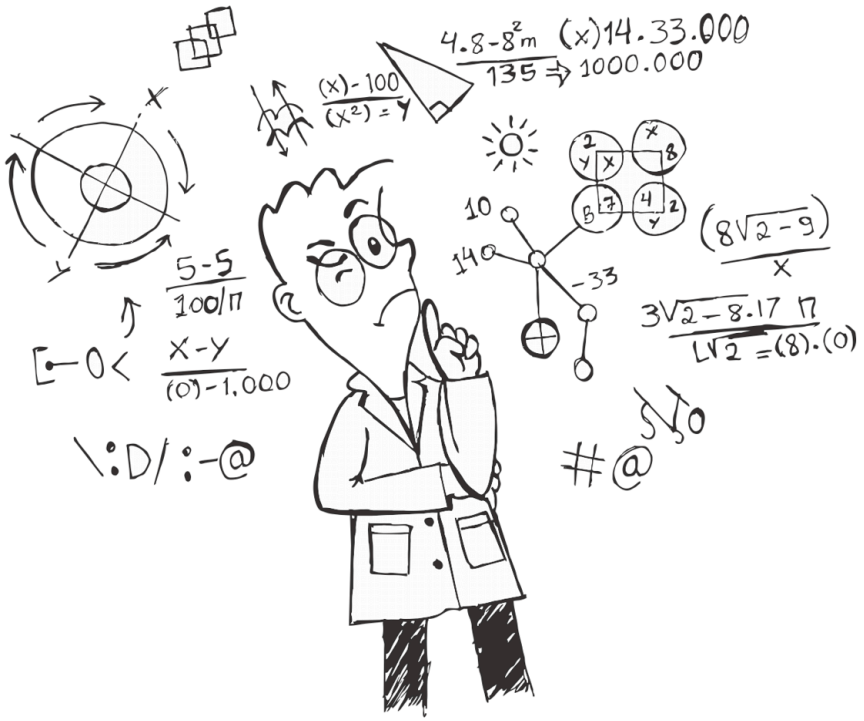

Figura 1.3: Códigos como testes de QI.

Velocidade de compreensão

Finalmente, o terceiro ponto e o que se pressupõe ser único. Aqui é aplicável tudo aquilo que é comumente discutido nas infinitas rodinhas de conversa entre programadores e o que você já deve estar cansado de ler nos últimos anos.

Com certeza, lemos nosso código muito mais do que o escrevemos. Faz sentido otimizar sua compreensão, o que resultará em menos tempo empregado no conjunto dessas duas atividades.

Em razão de a compreensão do código ser relativa a quem lê, muitos outros fatores podem influenciar na compreensão além da qualidade da escrita: familiaridade com a linguagem, padrões adotados pelo time, conhecimento sobre o assunto etc.

Não vou nem entrar no mérito sobre as **linguagens de programação esotérica**, como *INTERCAL* e *brainfuck* que, propositalmente, forçam todo código a ser incompreensível.

Ao codificar, você é um autor e artista — como um amigo relata em https://caiquerodrigues.me/a-arte-da-programacao —, e o seu trabalho será lido e apreciado. É seu papel assegurar que a leitura seja fácil.

Escrever códigos compreensíveis exige pensamento cuidadoso, o que se traduz em tempo. Às vezes, até para piorar um produto é necessário tempo, o tomate seco e a água com gás estão aí para confirmar isso.

Pior, manter o código compreensível requer muito esforço para seguir aquele princípio dos bons escoteiros: sempre deixar o acampamento em melhor estado do que como o encontrou.

Dar um capricho para agilizar a leitura requer um maior tempo de escrita - o que vai diretamente ao encontro do primeiro dos objetivos apresentados.

A qualidade como critério de avaliação

Para a avaliação de obras de arte, existem inúmeros fatores além do estético. Para começar, a beleza é uma característica subjetiva e varia conforme o avaliador. O que é belo para um pode não ser para o outro e poucos casos são unânimes – Gisele Bündchen, se você estiver lendo este livro, saiba que me refiro a você. Clara, minha amada esposa, a parte da Gisele é brincadeira, ok?

De que outra forma você explicaria alguém pagar milhões pela pintura *Blue Fool* de Christopher Wool, vendida por 5 milhões de dólares, que, literalmente, zomba de todos que a observam e ri da cara da sociedade?

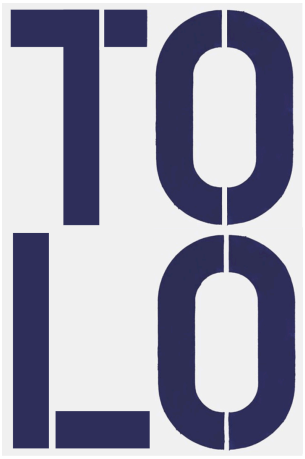

Figura 1.4: Imagem meramente ilustrativa que guarda fortes semelhanças com a original Blue Fool.

Fora a beleza subjetiva como critério de qualidade, cito também outros fatores que influenciam fortemente sua análise, como o impacto social, o contexto e a pressão exercida pela pessoa que paga seu salário, o famigerado chefe.

O código-fonte, enquanto produto de arte da computação, não é diferente. Pode e deve ser avaliado sob múltiplos aspectos.

Essas perspectivas nos levam a uma conclusão: existem momentos e lugares para os diferentes objetivos do produto.

> *Não há resposta certa, apenas diferentes sabores de errado.* - Robert Nystrom

Um sabor para cada propósito, complemento.

Assim como o código-fonte deve contemplar o seu propósito, a arquitetura do seu sistema deve atender ao objetivo que foi imposto. A regra é válida independentemente do nível de abstração, porque a qualidade e o propósito são qualidades indissociáveis, onde quer que sejam considerados.

Com cada vez mais conexões, mais dados e usuários mais exigentes, manter a responsividade do sistema é uma tarefa cada vez mais difícil. Por sorte, este é o propósito dos **sistemas reativos**: elasticidade e resiliência para a promoção da responsividade.

Considere-se um afortunado, pois esse é o tema do livro que você possui em mãos (ou no seu *hard drive*). Nele, abordaremos aquilo que você provavelmente já sabe e aplica no cotidiano, mas não costuma aplicar nos sistemas que desenvolve: a **perspectiva reativa**.

Aprenderemos a praticar o mantra do *deboísmo* no desenvolvimento de microsserviços: ficar de boa e reagir apenas quando realmente necessário. A introdução da perspectiva garantirá que, após a leitura, sua reação nunca mais seja a mesma ao escutar "a gente já chegou?".

1.3 Como ler o livro

Será de grande ajuda ao leitor ou à leitora entender que, assim como este capítulo, boa parte deste livro é sobre ponderações e

trade offs. Afinal, nem sempre é possível agradar a gregos e goianos...

Não se tratará do bom e do ruim, do certo ou do errado. Afirmações maniqueístas desse porte não costumam ser bem-vindas na Engenharia de software. No lugar dessa dicotomia explícita, o livro tem o despretensioso intuito de listar alternativas e destacar suas consequências. É um morde e assopra sem fim.

Em certas ocasiões, seja por experiência prévia ou por motivos religiosos, eu me posicionarei de maneira mais veemente. Especialmente quando o assunto for a adoção da reatividade. Tema pelo qual, como o título do livro sugere, nutro um profundo afeto e espero que, após a leitura, vocês também.

Vale ressaltar, no entanto, que essas ocasiões se tratam apenas da minha opinião e não devem ser tratadas, de modo algum, como mandamento bíblico. O livro não deverá ser lido como um conjunto de instruções literais. Cabe a você, exclusivamente, e mais uma vez, a tarefa de refletir e adotar a melhor opção dentro do cenário específico no qual se encontra.

Muitas vezes, a melhor opção será a própria experimentação. Tipo a crase: coloca, tira, coloca de novo e vê se está bonito. Se combinar com a frase, deixa e reza para que o resultado colabore.

Como autor e engenheiro de software, entendo que toda pessoa que lê tem seu próprio modelo mental e possui o direito de discordar dos pontos apresentados. É possível, inclusive, discordar de todos ao mesmo tempo – o que me faria repensar minha carreira e me lançar no ramo da dança.

Ainda assim, tenho a humilde pretensão de que este livro adicione mais um ponto a ser avaliado antes que a próxima decisão seja tomada na sua arquitetura de microsserviços. Desejo também que o livro sirva como munição para seu arsenal nas infindáveis rodinhas de conversas no café, ou, na pior das hipóteses, um inusitado suporte para o seu monitor.

Organização do livro

O livro é dividido em oito partes e cada parte é subdividida em capítulos, totalizando uns vinte e poucos capítulos.

Para ser sincero, os sistemas reativos poderiam ser um assunto para poucos capítulos. Contudo, um pequeno embasamento é necessário para que o assunto possa ser introduzido, o que justifica a adição desta primeira parte bem como da segunda (*Mais uma vez microsserviços...*).

A terceira parte (*Sistemas reativos*) apresenta a definição formal dos sistemas reativos, exemplifica e detalha dificuldades comuns advindas dessa abordagem.

As quatro partes seguintes (*Orientação a Mensagens*, *Elasticidade*, *Resiliência* e *Responsividade*) discutem e aprofundam as propriedades descritas no **Manifesto Reativo** e outros conceitos relacionados.

Por último, umas três ou quatro palavrinhas na parte *Considerações finais*.

Esses capítulos, em conjunto, fornecerão um bom panorama sobre o que é e, principalmente, sobre o que **não é** um sistema reativo. Ainda assim, ocasionalmente, haverá **reforços explícitos** para destacar e evitar que haja qualquer tipo de confusão.

Figura 1.5: Sistemas reativos, não confundir com sistemas reforçativos.

A disposição das partes do livro faz com que haja uma dependência de conceitos e que o livro seja, preferencialmente, lido de forma sequencial. A primeira parte antes da segunda, a segunda antes da terceira e assim vai.

Mas nada impede que alguém, durante um surto de rebeldia, o faça da forma que achar mais atraente. Não estou aqui para julgar ninguém, exceto quem gosta de pagode e pessoas que julgam outras baseadas em gosto musical.

Boa leitura

Se você gostou deste capítulo, nada garante que vai gostar do restante do livro. Ainda assim, dê um voto de confiança, esteja munido de saborosos biscoitos recheados e se deleite com a leitura.

Parte 2: Mais uma vez microsserviços...

Capítulo 2
Afinal, o que são microsserviços?

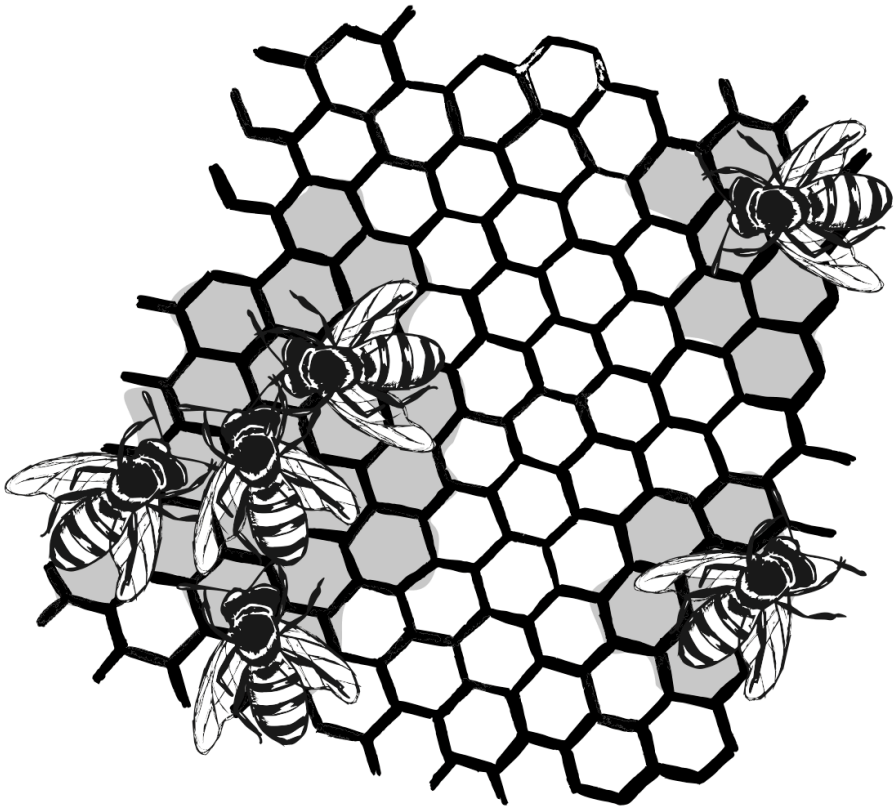

Figura 2.1: Uma colmeia.

Há anos, o termo microsserviços tem circulado incessantemente nas discussões de Engenharia de software. Ele é usado para descrever uma filosofia de *design* de arquitetura e, misteriosamente, vem sempre acompanhado de uma ilustração de colmeias de abelhas.

Bem-aventurado aquele que conseguiu passar ileso por essa avalanche de microsserviços e colmeias.

Apesar de hoje ser tão batido quanto as músicas da Simone no Natal, como comprova o gráfico a seguir, o movimento dos microsserviços só foi realmente impulsionado após o artigo de James Lewis e Martin Fowler (acessível em: https://martinfowler.com/articles/microservices.html), publicado em março de 2014, e, depois, pelo livro *Building Microservices: Designing Fine-Grained Systems* de Sam Newman.

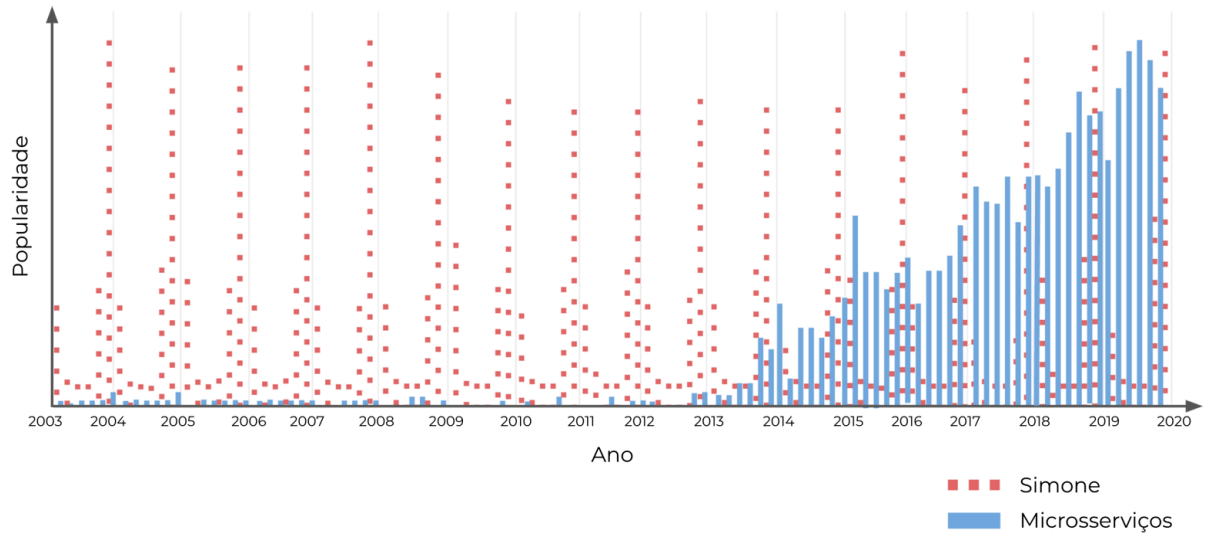

Figura 2.2: Popularidade do termo Microsserviços x Simone.

De lá para cá, enquanto os famosos *hits* da Simone se mantiveram apenas nos natais, os microsserviços galgaram rapidamente ao topo e lá estão até hoje – jamais desacompanhados de uma ilustração de uma colmeia de abelha – e desempenham um papel importante em muitas organizações. Embora não exista uma definição precisa, há características comuns entre as implementações que alegam utilizá-la.

2.1 Definição de microsserviços

A jornada da arquitetura de microsserviços começa com um monólito. Em um monólito, empacotamos nosso aplicativo inteiro como uma única unidade. Um grande pedaço de código, difícil de manter e escalar. A fim de lidar com os problemas relacionados aos monólitos, existe uma solução comum, ou seja, dividir o monólito no número de serviços independentes gerenciáveis e escaloná-los separadamente como e quando necessário. Esses serviços menores são conhecidos como microsserviços.

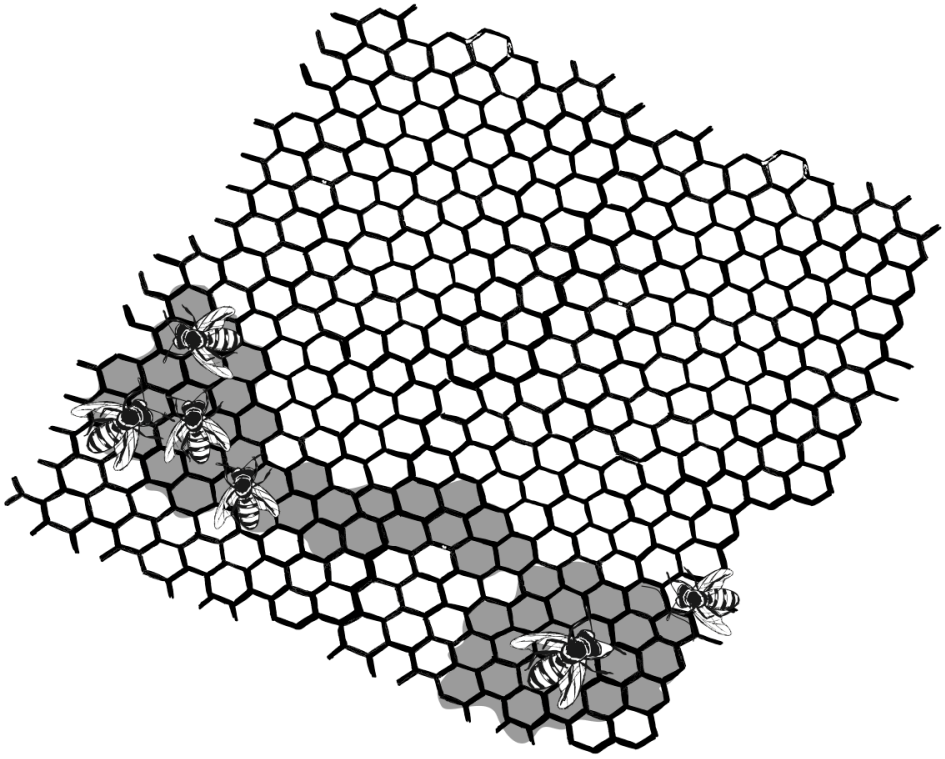

Figura 2.3: Outra colmeia.

Na minha opinião, um dos principais ganhos dessa arquitetura, não querendo desmerecer as demais, foi a introdução do termo monólito, que nada mais é do que uma pedra grande, ao nosso vocabulário para designar as aplicações que não são baseadas em

microsserviços. Provocando, por consequência, um caloroso embate sobre sua pronúncia correta.

A definição é fundamentalmente simples: distribuir uma aplicação monolítica em diversos pequenos serviços de forma que o conjunto deles entregue o valor final esperado. O tamanho e a orientação dessa distribuição são, propositalmente, aspectos imprecisos e nebulosos.

Não há nada que defina formalmente o tamanho do microsserviço. Há quem alegue que deve ter no máximo poucas dezenas de linhas de código, possivelmente por meio de *function as a service* (ou *FaaS*, do inglês: função como serviço). Outros alegam que o ideal é iniciar a partir de uma aplicação abrangente e, gradualmente, descobrir como esculpir os microsserviços.

Uma boa orientação para a questão de quando se deve dividir um microsserviço é a filosofia Unix de criação de utilitários modulares. Algo que se resume em: "faça uma coisa e faça bem".

Figura 2.4: Tamanho importa.

2.2 Justificativa

As leis de Lehman sobre a evolução de software ajudam a justificar o pensamento por trás do conceito. Dentre outras leis, evidencio, para fins didáticos: o **crescimento contínuo**, a **complexidade crescente** e a **conservação de familiaridade**:

- **Crescimento contínuo:** dado que a demanda é alterada após a concepção de uma funcionalidade, o software deve ser continuamente ampliado para manter a satisfação dos usuários, criando um círculo vicioso.
- **Complexidade crescente:** conforme o software é alterado, sua complexidade aumenta progressivamente.
- **Conservação de familiaridade:** a velocidade de evolução de um software está intimamente ligada ao grau de familiaridade dos profissionais que o mantêm.

Utilizando um pouco de lógica matemática, reordenando as sentenças e acrescentando uma pitadinha de sal, concluímos:

> O desenvolvimento e a manutenção de qualquer aplicação tendem ao caos e à gritaria.

Ou, como vi em um twit por aí:

> $e = mc^2$
>
> erro = mais × código²

Seja observando a combinação das leis anteriores, seja sofrendo na prática o que elas implicam, é questão de tempo para que o desenvolvimento de uma aplicação tenha dimensões colossais e que sua manutenção seja aterrorizante.

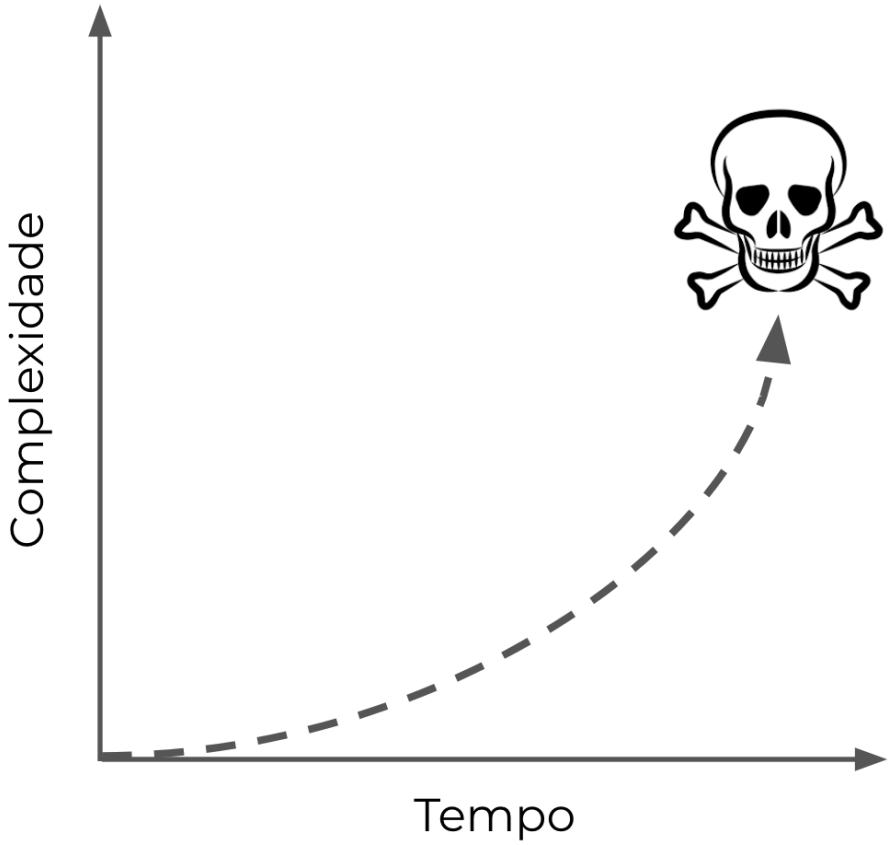

Figura 2.5: Curva exponencial da complexidade de manutenção de software em função do tempo.

É a inexorável segunda lei da termodinâmica aplicada no mundo do software. A desordem e a entropia só aumentam. É batata! Podem ser meses ou anos, mas o destino é certo: dedo no olho e correria. Uma alternativa é assegurar que as aplicações permaneçam micro e longe de dimensões estratosféricas, particionando-as de acordo com o domínio e conforme a diretriz acordada. Dessa forma, apesar de ainda tender ao apodrecimento, manter um software pequeno é menos custoso do que manter um maior.

Esse é o cerne da filosofia de microsserviços. Domínios distintos, na medida do possível, devem compreender serviços distintos. Busca-se, através da coreografia de minúsculos serviços distribuídos, a implementação das funcionalidades do sistema.

2.3 Conclusão

Assim como praticamente toda ideia apresentada no livro, a filosofia de microsserviços possui pontos positivos e negativos:

Prós da filosofia de microsserviços

- **Fronteiras de módulos bem definidas:** uma estrutura modular é favorecida e a lógica da aplicação está dispersa nos componentes. Apesar de uns serviços serem mais cruciais do que outros, a queda de um serviço individualmente não impede o funcionamento dos demais.
- **Entrega independente:** serviços simples são mais fáceis de serem entregues e, por conta da autonomia, é menos provável que causem falhas no ecossistema inteiro quando dão pane.
- **Diversidade tecnológica:** é possível utilizar e mesclar a linguagem, os frameworks e as ferramentas mais adequadas para o contexto. Isso deve ser salientado por promover a programação poliglota, os fluxos poliglotas e a persistência poliglota.
- **Reúso de serviços:** uma vez que o serviço é implementado e está testado, ele pode e deve ser reutilizado em outras soluções similares.
- **Resiliência:** a capacidade de o sistema se manter funcionando mesmo quando outro serviço está indisponível; é se recuperar graciosamente a partir da falha. Sistemas monolíticos não são resilientes e estão fadados ao fracasso quando uma funcionalidade que derruba o serviço é chamada. Já os sistemas reativos, como veremos a seguir, levam essa vantagem ao extremo; é como o Wolverine e a autocura.

Figura 2.6: Sistemas reativos, não confundir com sistemas regenerativos.

- **Evolução rápida:** a arquitetura possibilita acompanhar as rápidas mudanças do mercado. Esse ponto é, provavelmente, a maior vantagem da filosofia. Devido a ela, é possível construir times cross-funcionais, ou priorizar os serviços que agreguem maior valor. Pode-se também descentralizar as tomadas de decisão e minimizar a necessidade de coordenação de atividades bloqueantes.

Contras da filosofia de microsserviços

- **Consistência eventual:** a distribuição de dados faz com que a manutenção da consistência de informações em um sistema distribuído seja uma tarefa árdua. Todos os sistemas devem ser capazes de lidar com uma inconsistência eventual.

- **Acesso aos dados:** ao optar por uma arquitetura descentralizada, descentralizam-se também os dados de persistência. Logo, uma estratégia de gerenciamento de dados deve ser traçada para o acesso aos dados distribuídos. Determinadas estratégias, como veremos adiante, utilizam a duplicação de dados para solucionar esse problema.
- **Desempenho:** o desempenho do sistema pode ser afetado ao separar uma aplicação em diversas outras. Um dos muitos fatores que podem afetar tal característica, por exemplo, é a presença, antes inexistente, da rede e de sua característica estocástica.
- **Complexidade operacional:** por se tratarem de sistemas distribuídos, são complexos por natureza e demandam maturidade dos times para gerenciar múltiplos serviços e seus respectivos domínios e entregas.

Capítulo 3
Características dos microsserviços

Ao optar pela adoção de uma arquitetura baseada em microsserviços, apesar de não haver definição concreta sobre como realizar, existem características que ajudam a guiar e medir sua qualidade.

Essas características são relacionadas de modo que, ao aperfeiçoar uma, outra é aprimorada de bandeja e, ao falhar em alcançar alguma, outra é prejudicada. Esses itens são alvos de melhorias constantes e, não importa o estágio da arquitetura, sempre será possível melhorá-los. É como o tubo de pasta de dentes, não importa o quão usado esteja, sempre dá para extrair mais um pouco do conteúdo, é só questão de esforço e paciência.

Essa mudança de cultura de desenvolvimento exige que ferramentas e estratégias auxiliem o controle e facilitem o entendimento. Por exemplo, o uso de integração contínua, de testes integrados automatizados e de outras práticas que assegurem a qualidade é fundamental e imprescindível. Uma arquitetura de microsserviços deve buscar sempre aprimorar os seguintes pontos:

1. **Encapsulamento:** cada microsserviço é uma parte única de um quebra-cabeça maior que atende aos objetivos do negócio. Eles devem ser baseados no domínio, possuir requisitos funcionais e ser executados em um processo exclusivo, possivelmente hospedados em máquinas separadas de outros microsserviços.

Figura 3.1: Encapsulamento.

2. **Centralização na regra do negócio:** similar ao encapsulamento. Cada serviço deve alinhar seu contexto a um modelo de domínio para que haja sentido no ecossistema de microsserviços.

3. **Automação:** manutenção gerencial e suporte operacional eficientes. Empregar muitos serviços pequenos pode rapidamente se tornar um problema se a automação não for priorizada. Entrega automática, alertas integrados e testes automatizados são exigências indispensáveis.

4. **Descentralização:** cada serviço deve ser descentralizado e autônomo para que possa operar de acordo com sua própria agenda e prioridades. Cada serviço, se independente, pode ser atualizado e entregue a seu bel-prazer.

5. **Independência:** ser capaz de fazer o *deploy* individualmente. O sistema de software pode ser implantado como um todo, mas não é obrigatório. Os microsserviços podem ser adicionados e removidos do sistema durante a execução. Um microsserviço deve ser implantado com suas dependências, por exemplo, banco de dados e componentes terceiros.

6. **Tolerância a erro:** se um quebrar, outro não pode quebrar de modo imprevisível. Antecipação de falhas em tudo, junto com validação de entradas e validação de saídas. Uma falha não pode culminar em um efeito dominó – o jogo, não a banda.

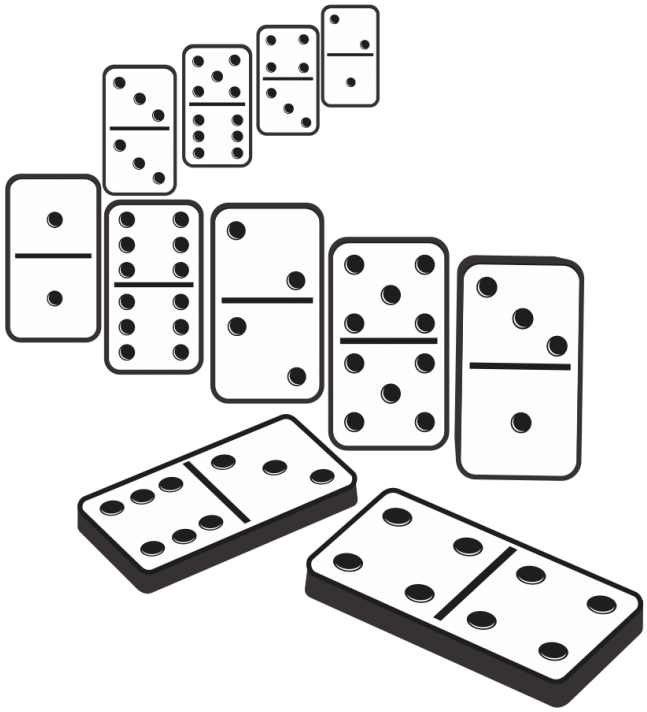

Figura 3.2: Tolerância a erros.

7. **Observabilidade:** a existência de múltiplos serviços possibilita escalabilidade e simplicidade de interação, todavia, para entender o panorama e evitar o caos generalizado, eles devem ser monitoráveis. Um padrão comum é agregar dados, métricas, *logs* e mensagens de todos os serviços e analisá-los com um suporte operacional centralizado. Alertas automatizados devem ser integrados de forma que, caso algum serviço esteja anômalo, a equipe de suporte e os desenvolvedores fiquem rapidamente cientes disso.

3.1 Autonomia, autonomia, autonomia e compensação

A comunicação entre os limites dos microsserviços é realmente um deus nos acuda. Dependendo de fatores como nível de acoplamento, quando ocorrer uma falha, o impacto no sistema poderá variar significativamente e beirar a catástrofe global.

Em um aplicativo baseado em microsserviços, com serviços distribuídos em vários servidores, ocasionalmente os componentes falharão. Falhas parciais e interrupções desastrosas certamente ocorrerão, portanto você precisa projetar seus microsserviços e a comunicação entre eles considerando os riscos dos sistemas distribuídos. É por isso que, se os microsserviços estiverem se comunicando por meio de cadeias dependentes, é possível argumentar que você tem um monólito distribuído, unindo o pior dos dois mundos. Para esses casos, duas palavras: "para" e "béns"!

Os microsserviços só são verdadeiramente independentes e desacoplados se puderem evoluir independentemente. Se você implementar uma cadeia de microsserviços vinculados por chamadas síncronas, quando um dos microsserviços falhar, toda a cadeia falhará. Um sistema baseado em microsserviços deve ser criado para continuar a trabalhar da melhor maneira possível durante falhas parciais.

Mesmo se você implementar uma lógica do cliente que use novas tentativas com intervalos exponenciais ou fluxos alternativos com *circuit breakers* e *fallbacks*, quanto mais complexa a cadeia de chamada síncrona for, mais complexa será a implementação de uma estratégia para falha, tendendo ao nível estratosférico de dificuldade de manutenção.

Outro ponto relevante dos fluxos secundários é sobre atualização e testes. Em sistemas que possuem muitas soluções alternativas, a tarefa de testar todas as combinações possíveis é impraticável e,

consequentemente, menosprezada. Isso resulta em soluções desatualizadas. Como esses fluxos são utilizados apenas em circunstâncias de erro, essa desatualização pode resultar em cenários piores do que o cenário do erro original, que supostamente eles deveriam solucionar.

Nesses casos, como pedem os sistemas reativos, replicação de dados em um serviço é uma boa solução. Sob outra perspectiva, a replicação de dados pode causar vazamento e obsolescência.

O que é mais benéfico? Replicar e manter os dados ou garantir que os serviços tenham uma disponibilidade alta o suficiente?

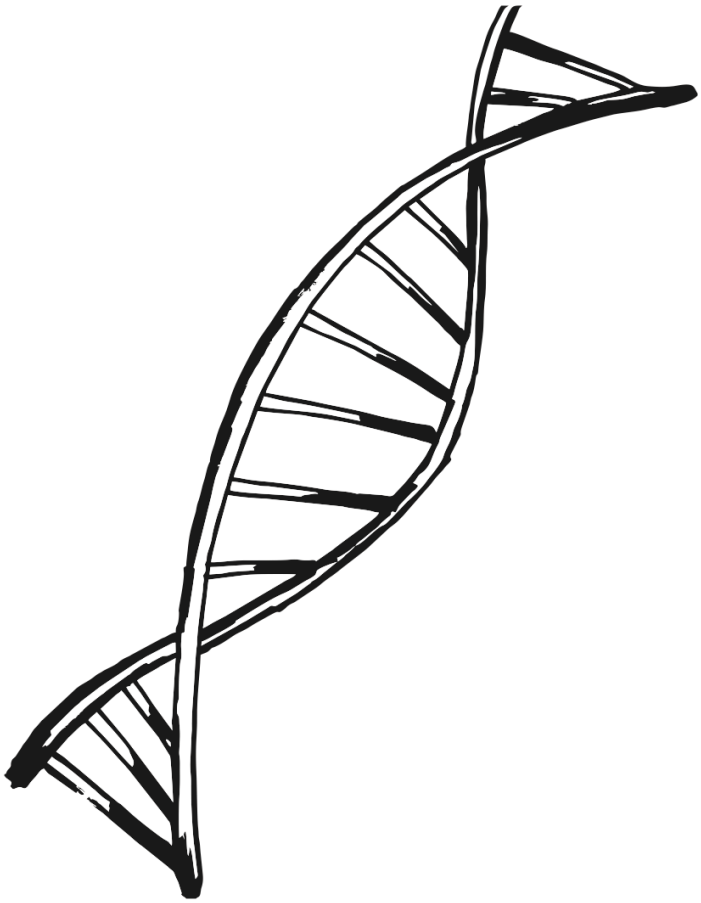

Figura 3.3: Sistemas reativos, não confundir com sistemas replicativos.

"A autonomia é supervalorizada. Os sistemas estão disponíveis em 99% do tempo. Alto para chuchu."

Disponibilidade total	Tempo de indisponibilidade por ano
99,999%	5 minutos
99,99%	52 minutos
99,9%	8,5 horas
99%	3,5 dias
90%	36,5 dias

Em um ano, a mesma disponibilidade de 99% representa um período doloroso de 3,5 dias. Você consegue imaginar o desespero do Mark Zuckerberg se o Instagram ficar fora do ar por 3 dias? Calma, que piora.

Matematicamente falando, a possibilidade de dez sistemas, cada um com 99% de disponibilidade, estarem disponíveis ao mesmo tempo seria 99^{10}%, cerca de 90%, ou 9.0438208e+19 para os amantes da precisão. Analisando a tabela anterior, percebemos que 90% de disponibilidade em um ano é o mesmo que 36,5 dias de indisponibilidade e a concretização do pior pesadelo do CEO da companhia.

Perceba que, com apenas esses dez serviços, o valor já é consideravelmente baixo. Adicione outras dezenas de microsserviços na equação e perceba a importância da autonomia.

Parece que o jogo virou, não é? É por isso que quem participa de projetos de larga escala tem que se acostumar com os rotineiros alarmes de acidentes e métricas indicando insucessos.

Pode ser que, em vez de melhorar a autonomia e o isolamento, faça mais sentido aumentar a disponibilidade dos serviços de 99% para

99,999%. Garantir largura de banda ou escalar os serviços pode ser considerado para resolver o problema.

O que nos leva, mais uma vez, a um *trade off*: autonomia e inconsistência vs. disponibilidade e alto custo?

3.2 Conclusão

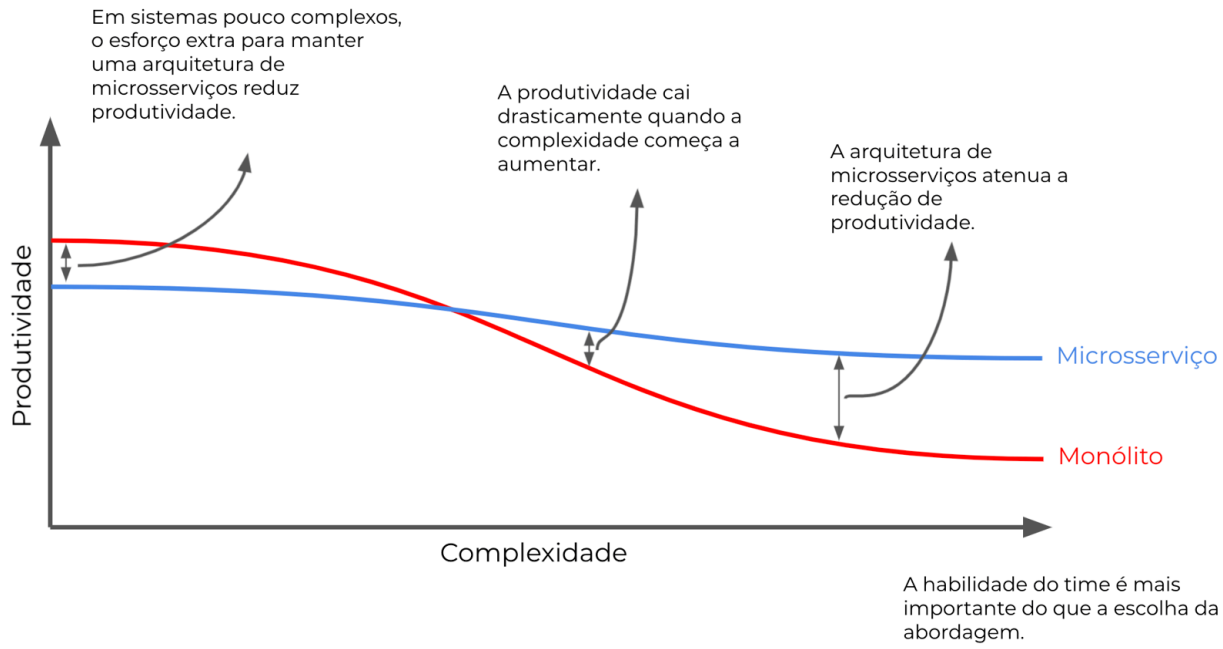

Figura 3.4: Produtividade dos microsserviços e dos monólitos pela complexidade do sistema.

A escolha sobre o uso de microsserviços nunca é fácil. Embora haja vantagens nítidas, também há um preço alto. O custo da produtividade na manutenção de uma arquitetura de microsserviços para sistemas pouco complexos, como afirma a ilustração anterior, é muito alto. Essa ponderação motivou, inclusive, a escrita do seguinte texto pela Rebecca Parsons (https://www.thoughtworks.com/pt/insights/blog/microservices-

adopt). Nele, ela advoga contra a adoção imediata dessa filosofia e afirma:

> *Nem todas as organizações estão preparadas para microsserviços*. - Rebecca Parsons.

O peso de cada característica varia de acordo com a estrutura organizacional da empresa, do domínio, da maturidade do time e de outros fatores de difícil mensuração. É fácil subestimar o esforço organizacional de colocar todas as equipes na mesma página e estabelecer uma comunicação confiável e eficaz. Não é incomum e nem inaceitável que a balança pese mais para a não utilização de microsserviços.

Como Sam Newman apontou durante sua apresentação na QCon Londres 2020 sobre padrões de decomposição de monólitos (https://www.infoq.com/presentations/microservices-principles-patterns/): "o monólito não é o inimigo" e "microsserviços não devem ser a escolha padrão".

No seu livro, *Monolith to Microservices*, Newman aconselha as pessoas a se concentrarem no resultado, não na tecnologia, e sempre lembrar que o objetivo é a implantação independente. No mesmo livro, Newman reitera ainda que a maioria das empresas se sairia melhor com a opção subestimada de um monólito modular do que com microsserviços.

No fim das contas, o resultado, a implantação e a familiaridade da equipe com a solução são mais importantes do que qualquer que seja a filosofia adotada.

Capítulo 4
Microsserviços e a Orientação a Objetos

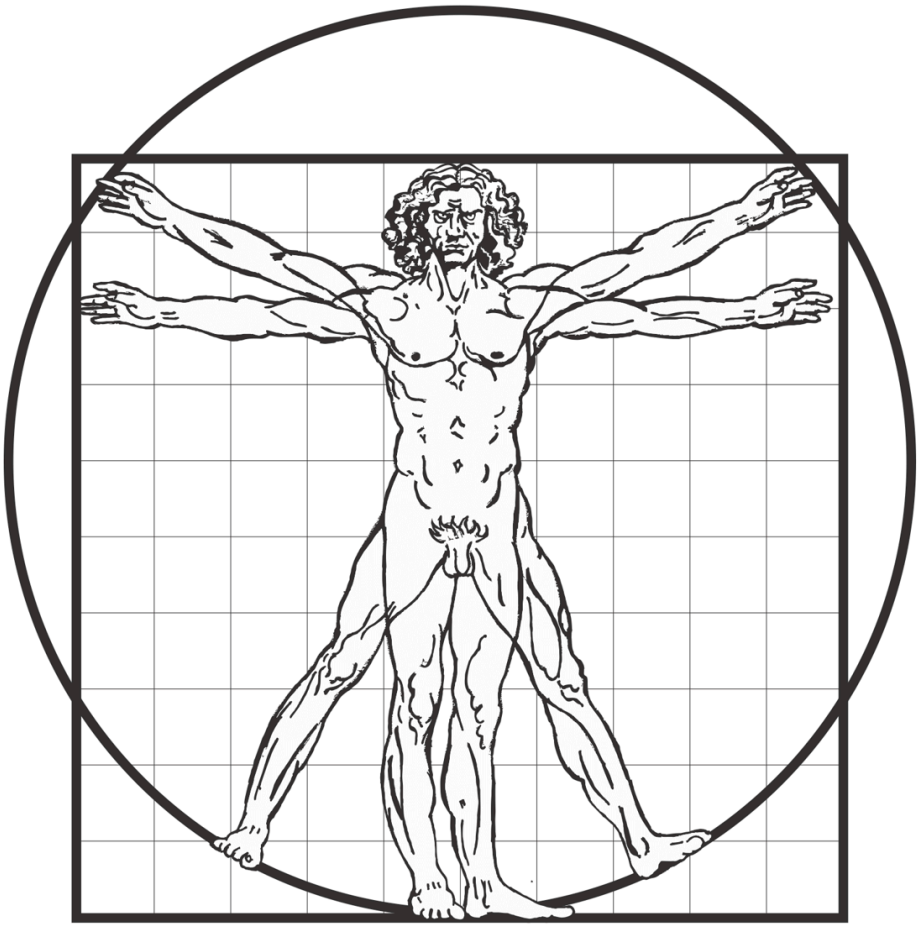

Figura 4.1: Plantas de construção

Neste ponto, você tem uma ideia sobre a aparência de um projeto de microsserviços de um sistema de software: vários componentes interagindo entre si por trocas de mensagens. Esses componentes devem ser independentes, possuir bom encapsulamento, ser centralizados nas regras de negócio e suas interações não devem ser fortemente acopladas.

Podemos reformular essa ideia e concluir que a arquitetura de microsserviços deve ter alta coesão e baixo acoplamento.

Se você está familiarizado com OO (Orientação a Objeto), deve ter notado que essa conclusão é compartilhada entre os dois paradigmas: OO e microsserviços. Projetos primorosos de OO possuem excelência em alta coesão e baixo acoplamento. Uma coincidência curiosa, para dizer o mínimo.

Em uma troca de e-mail, em 2003, Alan Kay, um dos "inventores" da OO, esclareceu o que ele pretendia quando cunhou o termo "Orientação a Objeto" (acessível em: http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en):

> *O paradigma de Orientação a Objeto, para mim, é apenas sobre mensagens, retenção local e proteção, ocultação de processos de estado e ligação tardia de todas as coisas*. - Alan Kay

Kay também alega:

> *Lamento ter cunhado o termo "objetos" há muito tempo para este tópico porque faz com que muitas pessoas se concentrem na ideia menor. A grande ideia é enviar mensagens*. - Alan Kay

Em outras palavras, de acordo com Alan Kay, os ingredientes essenciais para OO são passagens de mensagens e ligação tardia.

Observe agora um trecho de como Martin Fowler resumiu a arquitetura de microsserviços no já citado artigo *Microservices* (https://martinfowler.com/articles/microservices.html):

> *O estilo de arquitetura de microsserviço é uma abordagem para desenvolver um único aplicativo como um conjunto de pequenos serviços, cada um executando em seu próprio processo e se comunicando com mecanismos leves...* - Martin Fowler

Se destacarmos o denominador comum entre os trechos e tivermos um pouquinho de boa vontade, percebemos uma relação entre as definições de maneira que conseguimos traçar um paralelo: o paradigma de Orientação a Objeto e a arquitetura de microsserviços são, fundamentalmente, sobre encapsulamento e troca de mensagens.

Nesse paralelo, as classes estão para objetos assim como serviços estão para instâncias, e objetos estão para OO como instâncias estão para a arquitetura de microsserviços. Ambos são arquétipos de sua hierarquia, variando apenas no grau de abstração desejado porque se a OO é sobre o tijolo, os microsserviços são sobre casas e prédios.

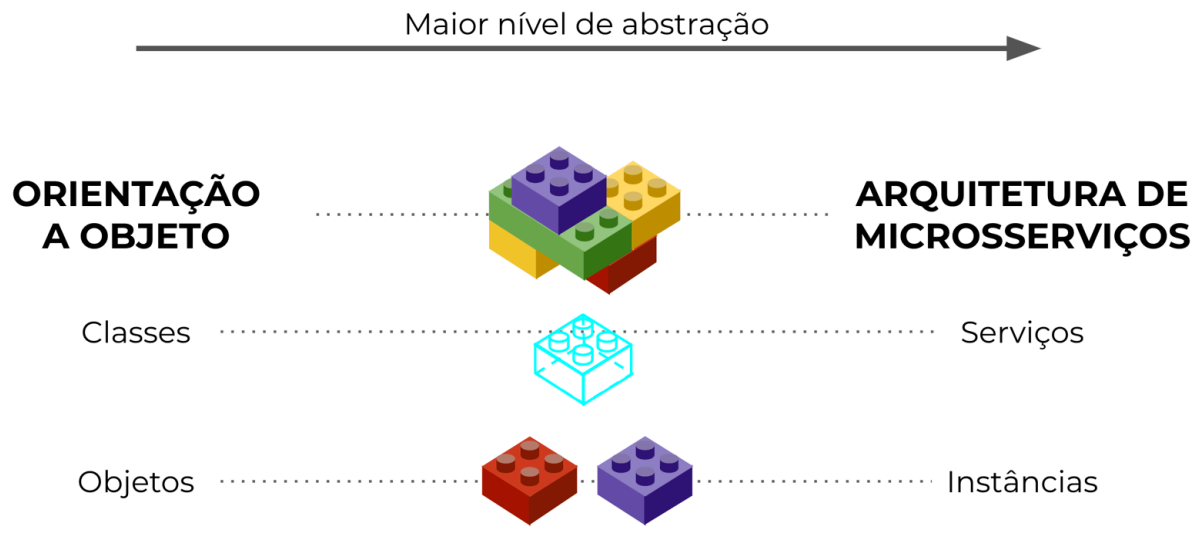

Figura 4.2: Analogia entre microsserviços e OO.

Algumas semelhanças são bem intuitivas: ambos são conceitos baseados em um ecossistema composto por *building blocks* trabalhando em conjunto. Os dois dão grande importância ao encapsulamento dos dados e ao grafo de dependências de transações de informações via chamadas a outros componentes. Apesar das propriedades intrínsecas em cada conceito que evitam uma aproximação maior, há uma interseção, em distintos níveis hierárquicos, e um paralelo entre os dois é completamente viável.

A facilidade de comunicação entre objetos na OO permite um design mais intuitivo. A ausência dessa proximidade nos microsserviços, por outro lado, nos obriga a ter uma maior cautela. Microsserviços geralmente rodam em máquinas diferentes, exigindo, portanto, a presença de uma rede de comunicação e tratamentos inerentes, como **perdas de pacote**, **retentativas** e **erros de integração**.

Outros problemas decorrentes da segregação das máquinas são a latência, que cresce com a profundidade das dependências, o acoplamento temporal, falhas em cascata, orquestração complexa e inconsistente, proliferação de armazenamento e fragmentação de dados e dificuldade em definir o estado do sistema em um determinado ponto.

A Orientação a Objetos, diferentemente dos microsserviços, possui normas e princípios conhecidos há décadas.

4.1 Princípios de OO nos microsserviços

Proponho uma releitura de outros princípios no universo OO sob a ótica da arquitetura de microsserviços. Pretendo usar o poder das releituras para o bem, diferentemente do que a Paula Fernandes e o Luan Santana fizeram com a música *Shallow* da Lady Gaga.

Por exemplo, o princípio ***DRY*** (*don't repeat yourself*, em português, não repita a si mesmo), corriqueiro na OO, pode ser reinterpretado

sob a perspectiva dos microsserviços da seguinte maneira: ao perceber uma funcionalidade comum entre microsserviços, uma análise deve ser feita para uma possível abstração em um código compartilhado por ambos ou na criação de um terceiro microsserviço.

Aspiro demonstrar que outros princípios consolidados de OO e nossa experiência adquirida sobre eles podem servir de guia para desenhar uma arquitetura de microsserviços limpa e saudável. Fazer uso de um caminho das pedras provado e maduro enquanto projetamos um sistema é, sem dúvida, um *tchan* a mais.

Figura 4.3: Mais uma colmeia.

Princípios de OO

Um pouco após a OO se tornar popular, no início da década de 90 do século passado, também se tornou claro que a Orientação a Objeto não seria à prova de erros humanos.

Classes e métodos confusos, abstrações malfeitas e outras bizarrices continuavam possíveis, um verdadeiro ninho de mafagafo. Aparentemente, mudar o paradigma para OO não seria suficiente para resolver o pandemônio em que a programação se encontrava. Não que esteja muito diferente hoje em dia.

Meu ponto é que, embora a OO tenha vindo para melhorar, ainda era possível escrever código ruim.

Por conta desse cenário apocalíptico e para tentar resolvê-lo, vários princípios de design surgiram. Além dos já citados "alta coesão e baixo acoplamento" e *DRY*, é comum encontrarmos na literatura termos, como **Lei de Demeter**, ***YAGNI*** (*you ain't gonna need it*, você não vai precisar disso) e ***Tell, don't ask*** (Diga, não peça). Por julgar possuir maior importância e representatividade, dedicarei a próxima sessão para um outro termo, o ***SOLID***.

O que aconteceria se utilizássemos os conceitos dos princípios SOLID na arquitetura de microsserviços?

4.2 Princípios SOLID nos microsserviços

Figura 4.4: Sólido como uma pedra.

SOLID é um conjunto de cinco princípios de design que guiam a implementação de códigos Orientados a Objetos. Desenvolvedores(as) usam-nos como regras para implementar sistemas com linguagens OO. Este conjunto de princípios se tornou conhecido pelo Uncle Bob e cada letra do acrônimo representa um princípio, do inglês: ***Single Responsibility*** (responsabilidade única); ***Open/Closed*** (aberto/fechado); ***Liskov Substitution*** (substituição de Liskov); ***Interface Segregation*** (segregação de interface) e ***Dependency Inversion*** (inversão de dependência).

Apesar de não haver garantias de que será à prova de balas, quando contemplando inteiramente esses princípios, projetos bem-sucedidos costumam utilizá-los. Funcionam até comigo.

Existem toneladas de artigos descrevendo em que consiste o princípio de design SOLID. Por isso, não pretendo ruminar esse assunto e focar, mais uma vez, no significado dos termos e como se aplicam à OO. Falarei como os termos podem ser interpretados sob a ótica dos microsserviços, princípio por princípio, e qual ganho isso pode trazer.

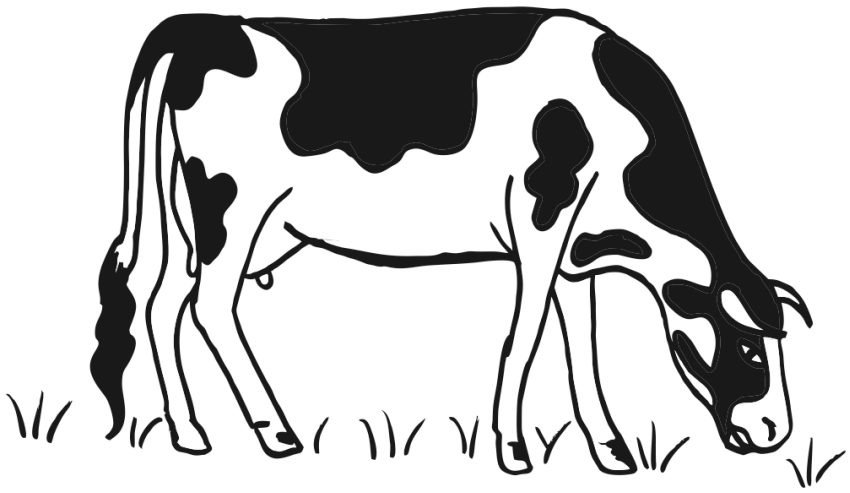

Figura 4.5: Sistemas reativos, não confundir com sistemas ruminativos.

Princípio da responsabilidade única

O componente deve ter um, e apenas um, motivo para mudança. Se ele for modular o suficiente, será possível reutilizá-lo em múltiplos casos. O lendário artesão de software Robert Martin, ou Uncle Bob, no livro *Arquitetura Limpa: O Guia do Artesão para Estrutura e Design de Software*, explicita a aplicabilidade desse princípio em outros níveis de abstração com outros nomes: *Common Closure Principle* e *Axis of Change*. Esses princípios, ainda segundo Martin, podem ser resumidos pela seguinte sentença: reúna coisas que mudam juntas e pela mesma razão e separe as que mudam em momentos diferentes e por motivos diferentes.

Princípio do aberto/fechado

Um microsserviço nunca deve ter que ser modificado para fornecer funcionalidades de caso extremo (que não respeitem o domínio) apenas pela conveniência. Em vez disso, para tratar tais cenários, os microsserviços devem ser facilmente estendidos e integrados a outro microsserviço, o que incentiva a composição de domínios.

Princípio da substituição Liskov

Uma nova versão de um microsserviço deve sempre ser capaz de substituir uma anterior sem quebrar nada. De modo geral, qualquer mudança que **exija atualizações de serviços terceiros** é desaconselhada. A presença do verbo "exigir" desempenha um papel determinante na afirmação. Para tal, é recomendada a prática de versionamento de interfaces de comunicação, como *endpoints* http e filas.

Princípio da segregação de interface

Um microsserviço não deve expor métodos que não estejam diretamente relacionados ao seu domínio. Ao expô-los, você não poderá mais mudar o microsserviço facilmente ou quebrará o princípio da *substituição de Liskov*, portanto estará "preso" em um domínio que não pertence ao microsserviço, quebrando também o princípio da *responsabilidade única*. Para sintetizar, podemos reescrever da seguinte forma: não permita a dependência de coisas que não são necessárias.

Princípio da inversão de dependência

Um microsserviço não deve depender de outros microsserviços. Deve depender de outros domínios que, por sua vez, são de responsabilidade de outros microsserviços. O ideal é que eles não tenham sequer ciência da existência uns dos outros. Assim, o acoplamento é mais fraco e, caso desejemos expandir o grau de abstração do domínio particionando o microsserviço em dois ou mais, a alteração terá menos complicações, favorecendo a autonomia de cada microsserviço.

4.3 Conclusão

O livro do Uncle Bob, mencionado na releitura do princípio da *responsabilidade única*, chega a resultados similares e traz outros excelentes tópicos sobre a organização da estrutura de uma aplicação. Recomendo a leitura.

Ao aplicarmos SOLID e outros princípios de OO como um norte para o design de microsserviços, obteremos uma melhor arquitetura. Obedecer aos princípios SOLID é muito mais simples do que respeitar diretrizes nebulosas e definitivamente melhor do que não respeitar nada.

Se aplicados corretamente, os princípios promoverão as características dos serviços que vimos nos parágrafos anteriores: serão modulares, combináveis, estáveis, transparentes e reutilizáveis. Assim, a arquitetura será mais sustentável, confiável, eficiente e utilizável.

Seguir os princípios é, na verdade, uma consequência direta de microsserviços bem projetados. Em contrapartida, uma arquitetura que não obedece aos princípios SOLID é, para poupar nos termos, desajeitada.

Figura 4.6: Onde é o agreste?

Na ausência de princípios claros para a construção de microsserviços, apesar de existirem outras maneiras, eu diria que os princípios da Orientação a Objeto são um excelente ponto de partida.

Parte 3: Sistemas reativos

Capítulo 5
Um discurso de ódio à proatividade

Figura 5.1: A gente já chegou?

Em Shrek 2, em uma cena tão irritante quanto clássica, o Burro Falante mostra uma de suas melhores facetas, a de ser uma desagradável companhia. Durante uma viagem extrovertida e animada, o equino começa a perguntar, de modo repetitivo, "A gente já chegou?" ao Shrek, seu companheiro de viagem. O animal tagarela está ansioso para chegar ao reino Tão Tão Distante e não consegue conter a empolgação. A frequência da indagação e a inquietação do Burro aborrecem o ogro esverdeado e fazem com que ele esbraveje com o amigo.

Honestamente, não lembro bem como o Shrek responde, mas, reforçando o estereótipo rabugento e ignorante dos ogros, presumo que não tenha sido inteligente e, muito menos, delicado.

De maneira quase unânime e intuitiva, sabemos que a resposta ideal seria algo natural e sagaz como: "Ainda não, aviso quando chegarmos". Essa resposta seria inteligente, porque todos ganhariam nessa situação: o Burro saberia quando chegasse, e o Shrek poderia apreciar seu sossego calando o desagradável companheiro de viagem.

Parece razoável, não é? Por que, então, apesar de reconhecermos que o Burro não está sendo esperto, replicamos esse comportamento na nossa arquitetura de microsserviços?

Trabalhemos em cima do hipotético, contudo não incomum, cenário.

Não incomum cenário

Em um *e-commerce*, porque todo exemplo exige um *e-commerce*, construído sob a filosofia de microsserviços, temos um serviço para encapsular o domínio de preços e outro para tratar de produtos.

Figura 5.2: Apenas TRÊS REAIS.

Por haver uma íntima relação entre os domínios, e uma supervalorização de http, como veremos na parte *Orientação a Mensagens*, é comum pensar em expor um *endpoint* na aplicação de

preços em que, dado um determinado identificador de produto, o preço é retornado.

Assim, sempre que o serviço de produtos precisar recuperar o valor de uma dose de café, o serviço de preços estará à disposição para atendê-lo.

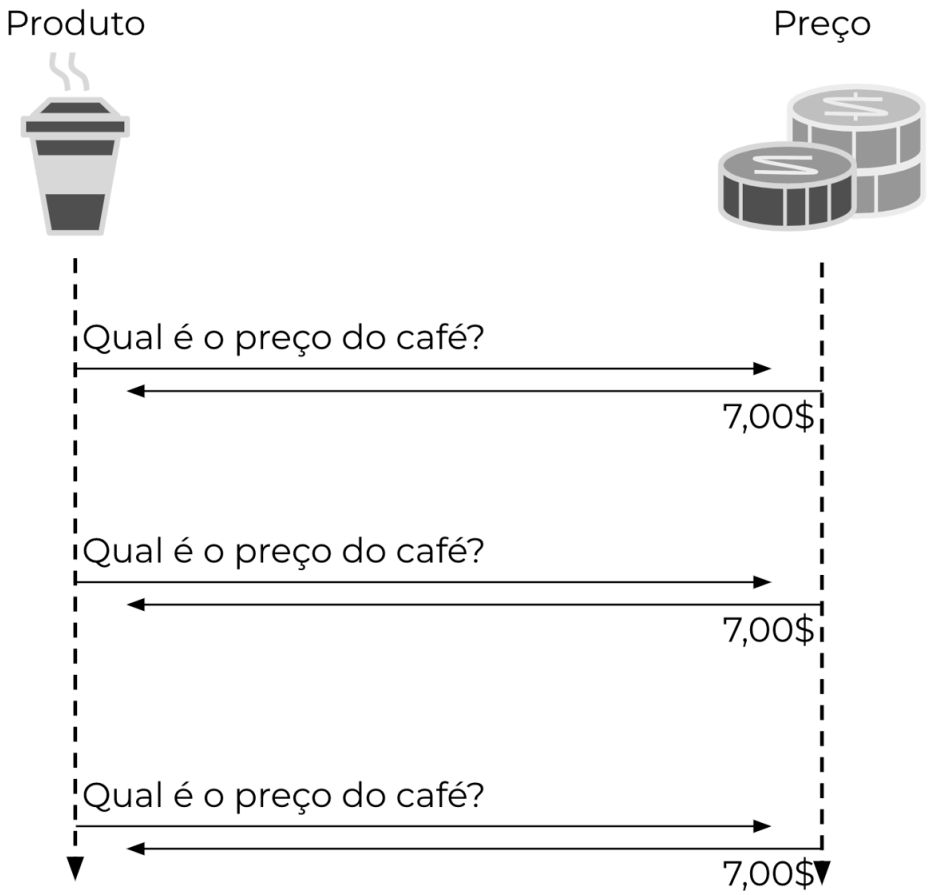

Figura 5.3: "A gente já chegou?", com outras palavras.

E isso pode acarretar uma indagação frequente, consumindo recursos e tirando a paciência do sistema proprietário do dado.

Notaram alguma semelhança?

Pausa dramática para reflexão...

Figura 5.4: A gente já chegou?

O problema consiste em, assim como o sr. Burro Falante, o serviço de produtos tentar recuperar informação proativamente. Reflitamos, o que aconteceria se o sistema detentor da informação, serviço de preços/Shrek, estivesse indisponível/batido as botas? Há uma certa ingenuidade e falta de noção em esperar a eterna disponibilidade e boa vontade ao perturbar a paz de espírito de um terceiro para obter a informação.

Imagine se existissem duas instâncias de Burro Falante desejando a mesma informação e atazanando o juízo do paciente Shrek. O que aconteceria se multiplicássemos o Burro Falante para várias instâncias da mesma forma como os *coaches* brotaram nos últimos anos? Imagine se outro animal/serviço também precisasse dessa

informação? O mundo sucumbiria! Coitado do Shrek e do serviço de preços... O escalonamento hipotético desse cenário nos ajuda a perceber que essa não é a melhor solução.

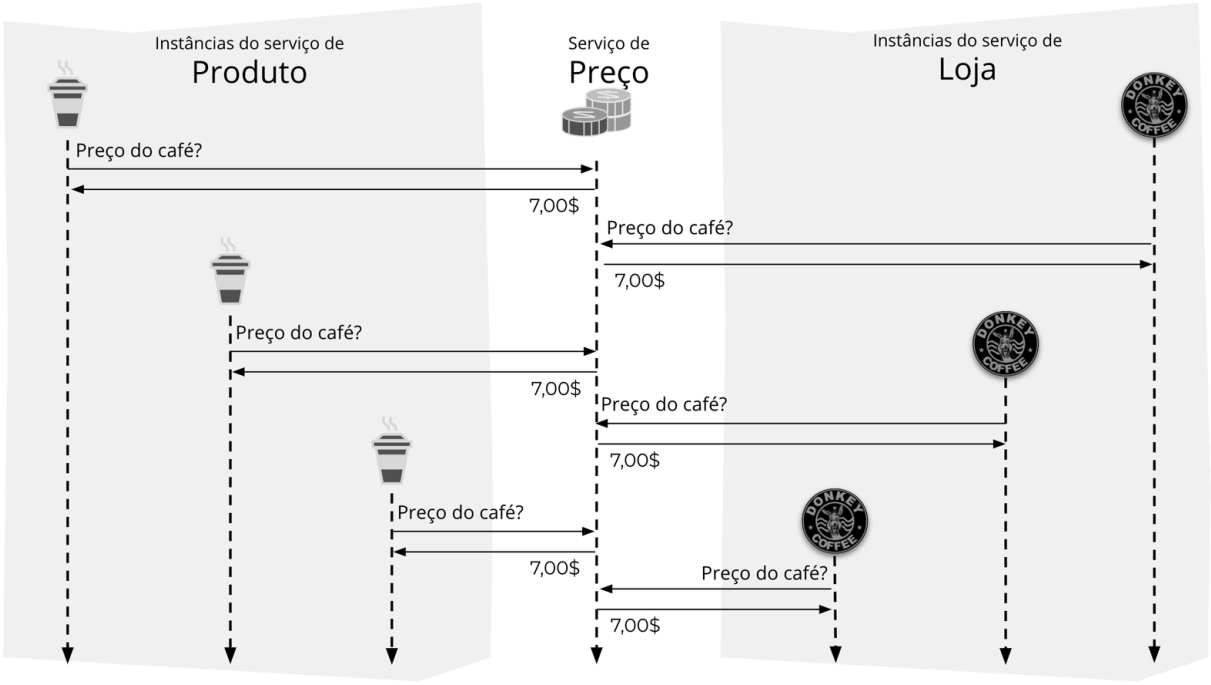

Figura 5.5: Escalonando mais do que coach quântico.

Estou cada vez mais tentado a acreditar que, assim como dirigir um carro por redondezas perigosas, solicitar informação nem sempre é uma boa ideia.

5.1 Diga, não peça

Eu não sou o primeiro a pensar assim. *Tell, don't ask* é um princípio, já mencionado neste livro, que ajuda as pessoas a lembrar que a Orientação a Objeto é sobre encapsular dados e operar sobre eles.

Como vimos anteriormente, muitos princípios da OO extrapolam o propósito e podem ser aplicados na arquitetura de microsserviços.

Melhor do que requisitar dados a um objeto, é instruir o objeto sobre o que fazer com eles, promovendo o encapsulamento.

Método GET do http

Confesso que tenho dificuldade na aceitação da utilização desse método http. Sem querer dar muito *spoiler* sobre a parte de *Orientação a Mensagens*, adianto que não consigo ver forma adequada de usá-lo. Salvo quando há exigências, tais como um *browser*, aplicativos que se comunicam com serviços *front end*, *gateways*, serviços terceiros que só consomem esse método etc. Por outro lado, expor voluntariamente um recurso via *GET* sempre me faz repensar várias vezes se estou certo.

Além de ferir o princípio de OO *Tell, don't ask*, o método *GET* ressalta uma das duas dificuldades da computação, que é, segundo Martin Fowler (link do artigo ao final do parágrafo), a de nos expressar de maneira concisa. Afinal de contas, qual o significado real da palavra "*get*"? Segundo a edição de 1989 do *Oxford English Dictionary* (foi a mais atualizada que encontrei pelo Google), há 289 significados, figurando em sexto lugar no *ranking* de palavras do inglês com mais sentidos. Por curiosidade, a primeira posição dessa competição é outra palavra queridinha da computação: "set", com 430 definições. Link do artigo de Martin Fowler: https://martinfowler.com/bliki/TwoHardThings.html.

Figura 5.6: http GET?

Cache, para que te quero?

Solicitar uma informação não é tão ruim se a resposta da solicitação for sempre a mesma. É como se o Shrek anotasse "não" em um papel e, sempre que o Burro Falante perguntasse, o papel com a resposta seria exibido pelo ogro. Isso economizaria tempo e raciocínio.

Caso o Shrek entregasse o pedaço de papel para o Burro falante para que, em vez de perguntar, ele o lesse sempre – e o Burro seguisse essa ordem, claro –, o Shrek conseguiria, enfim, seu merecido sossego. O sorridente equino seria capaz de responder sua própria pergunta antes mesmo que ela fosse feita.

Analogamente, o serviço dos produtos armazenaria a informação do preço em um banco de dados local para otimizar o desempenho em uma futura busca, ou seja, faria **réplica** dos dados em *cache*. Em um cenário em que múltiplas instâncias do mesmo serviço

coexistem, todas elas têm acesso à camada de persistência da base de dados compartilhada entre elas, como pede a cartilha dos microsserviços. Dessa forma, essas instâncias nem sequer necessitariam buscar a informação no serviço detentor dela, poupando troca de dados e potenciais ruídos de comunicação.

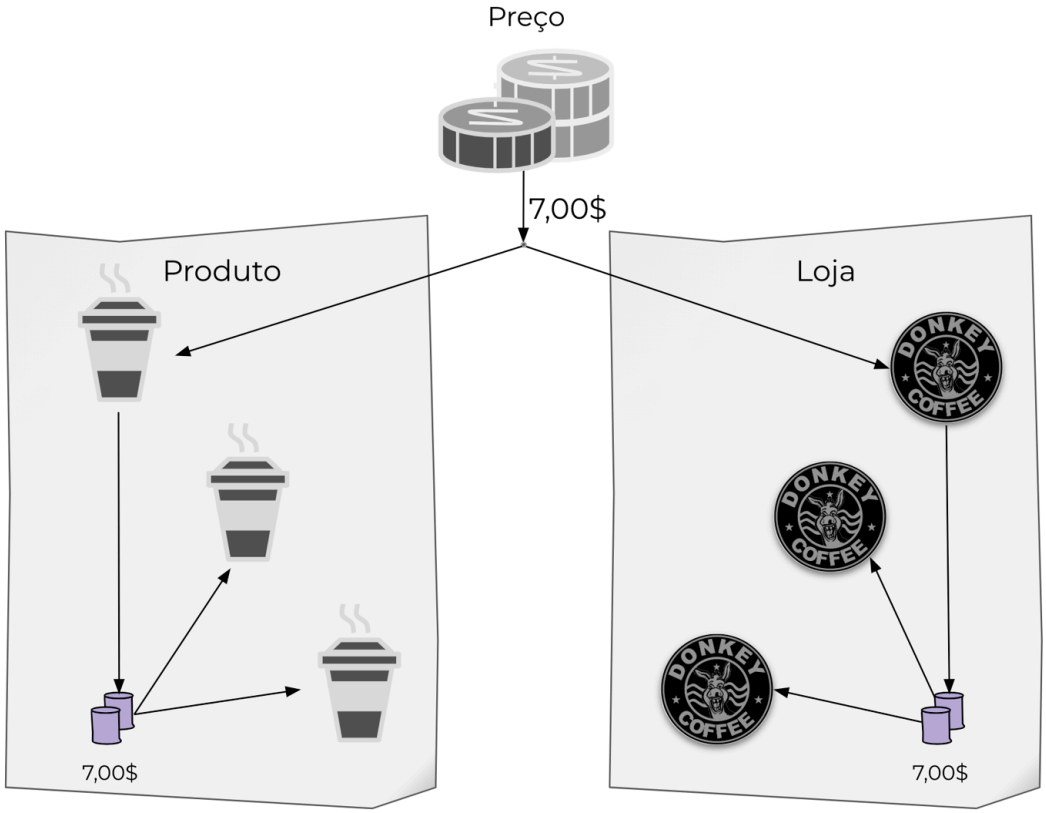

Figura 5.7: "A gente já chegou?" com cache.

A invalidação do cache, entretanto, não é trivial, porque aparentemente o universo não quer que você seja feliz. Ainda segundo Martin Fowler e seu texto sobre dificuldades da computação, a invalidação do cache é a outra grande dificuldade da computação.

Em que momento o Shrek atualizaria a resposta e daria outro pedaço de papel ao Burro?

Uma alternativa para facilitarmos a invalidação do cache é aceitá-lo como válido e atualizá-lo quando explicitamente ordenado. Em outras palavras, quando chegassem ao destino, o Shrek entregaria outro pedaço de papel com a palavra "sim" escrita.

O cache seja louvado

Em um cache em que informação nunca é inválida, o dado estaria sempre acessível sem precisar da consulta em outro sistema. Poupando qualquer participação do Shrek e do serviço de preço.

Se o microsserviço de produtos já possuísse o preço da bendita dose de café, esse dado não precisaria ser capturado por outro sistema, favorecendo a autonomia e enfraquecendo o acoplamento. Pontos extremamente positivos e chave em uma arquitetura de microsserviços.

Isso nos leva a outros dois questionamentos: **como obteríamos a informação inicial** e **como atualizaríamos esse valor?**

A resposta para as duas perguntas é a mesma: o sistema de preços diz o preço dos produtos sempre que houver uma adição ou alteração de preço. O Shrek entregaria outro papel ao Burro sempre que o endereço de Tão Tão Distante mudasse. O serviço de produtos e o Burro reagiriam ao envio da alteração e replicam essa informação, adaptando-a às suas óticas.

Como se não fosse suficiente, ainda pouparíamos a mensagem que consulta os dados. A própria fala: "A gente já chegou?" não existiria mais. Isso evitaria gastos desnecessários e melhoraria o desempenho.

Figura 5.8: Um novo mundo.

5.2 Conclusão

Percebem aonde chegamos? Não estou falando do reino de Tão Tão Distante. Nós chegamos à resposta que o Shrek deveria ter dado. Aquela que julgamos sagaz e natural, indicada no início do capítulo: "Ainda não. Aviso quando chegarmos" é uma resposta que carregamos conosco há anos e, apesar de ser simples, não costumamos aplicar nos nossos sistemas.

Para chegar a essa solução, tivemos que mudar a perspectiva com a qual vemos o cenário: a perspectiva reativa. Na qual toda – ou quase toda – proatividade é devidamente punida.

Resumindo, para alcançarmos a reatividade, tivemos que parar de, proativamente, importunar outro sistema e passar a reagir ao evento de alteração do valor no qual estou interessado. De quebra, além de sossegarmos o facho e deixarmos o outro sistema em paz, ganhamos uma maior autonomia, enfraquecemos o acoplamento e evitamos a requisição que solicita uma resposta. Só vi vantagens.

Figura 5.9: Sistemas reativos, não confundir com sistemas receptivos.

Todavia, como um dos contras dessa abordagem, temos também a consistência debilitada.

O que acontece se o microsserviço de preços atualizar o valor do café e essa informação não for repassada ao sistema de produtos em tempo hábil? O que aconteceria se Tão Tão Distante mudasse de localização e o Shrek não pudesse comunicar isso ao Burro Falante? A informação obsoleta poderia acarretar uma série de problemas, desde prejuízos financeiros até Burros Falantes vagando por Quixeramobim (CE) pedindo informações.

Eventualmente, em um futuro não Tão Tão Distante, na parte *Elasticidade*, tornarei a falar sobre isso. *Spoiler*: a inconsistência é inevitável e a **consistência é uma ilusão** tão grande quanto, sem querer arruinar a experiência de ninguém, a ilusão de que tocar bem no Guitar Hero o torna um bom guitarrista na vida real.

Capítulo 6
A Reatividade diz "olar"

Figura 6.1: Senhoras e senhores, sejam bem-vindos ao mundo da reatividade.

Se você pesquisar por "sistemas reativos", encontrará resultados bem acompanhados por termos, como **microsserviços**, **sistemas orientados a eventos**, ***event sourcing*** e incontáveis outros similares.

Muito embora não sejam a mesma coisa, eles andam de mãos dadas na indústria de desenvolvimento e costumam ser sobre aplicações distribuídas e sobre suprir a necessidade que o mundo atual exige.

Discussões entre esses termos e sua adoção costumam ser mais acaloradas do que as discussões sobre se o Ross e a Rachel estavam

dando um tempo, em *Friends*. A verdade é que os requisitos mudaram, os usuários estão mais exigentes e o casal estava sim dando um tempo no *sitcom*.

As demandas se tornaram mais rigorosas. As requisições, assim como fizeram os milhões de dados que transitam nelas, se multiplicaram como coelhos depois de algumas doses de catuaba. É natural que a arquitetura de sistemas da solução mude junto.

É necessário ser responsivo, independentemente da técnica que se utilize. As práticas que adotávamos há alguns anos e eram perfeitamente razoáveis já não são suficientes para dar ao usuário uma boa experiência. Assim como, provavelmente, as práticas de hoje não serão adequadas para o mundo em que viveremos – se deus quiser – daqui a 10 anos.

Os dados precisam estar disponíveis em todas as circunstâncias, servindo aos usuários e às máquinas em tudo que é fuso horário, continuamente e em tempo quase real.

As arquiteturas, técnicas e ferramentas tradicionais se provaram não responsivas, não escaláveis e não disponíveis. As empresas se movimentam em uma direção diferente visando fornecer uma enorme quantidade de dados rapidamente de uma forma preditiva, elástica e resiliente.

Uma galera massa, ao perceber o que estava acontecendo, resolveu juntar tudo e colocar no papel. Segundo eles, todos os aspectos necessários para haver uma abordagem consistente para arquitetura de sistemas já eram bem conhecidos. Nasciam então os sistemas reativos.

Essencialmente, essa junção resultou no Manifesto Reativo (disponível em: https://www.reactivemanifesto.org/pt-BR). Apesar do meu desgosto pessoal por manifestos no geral por considerar que eles clamam por atenção e da tendência de parecer com apresentações do PowerPoint, devo admitir que este ficou bem bacana.

Manifesto Reativo

Sistemas criados como reativos são muito mais flexíveis, desacoplados e escaláveis. Isso os torna mais fáceis para desenvolver e mais abertos a mudanças.
São significativamente mais tolerantes às falhas e, quando elas ocorrem, são tratadas com elegância em vez de desastre.
Sistemas reativos são altamente responsivos, dando aos usuários um efetivo feedback interativo.

Os sistemas reativos têm uma visão holística do design do sistema, focando em manter os sistemas distribuídos responsivos, tornando-os resilientes e elásticos.

Podemos afirmar que sistemas reativos são sobre a passagem de mensagens assíncronas. Por isso, permitem o desacoplamento temporal entre sistemas e promovem a responsividade em falhas e sob extremas demandas. Soa algo como: microsserviços de uma forma bem-feita, não é?

O manifesto apresenta uma ilustração. Tomei a liberdade de estilizá-la e apresentá-la a seguir.

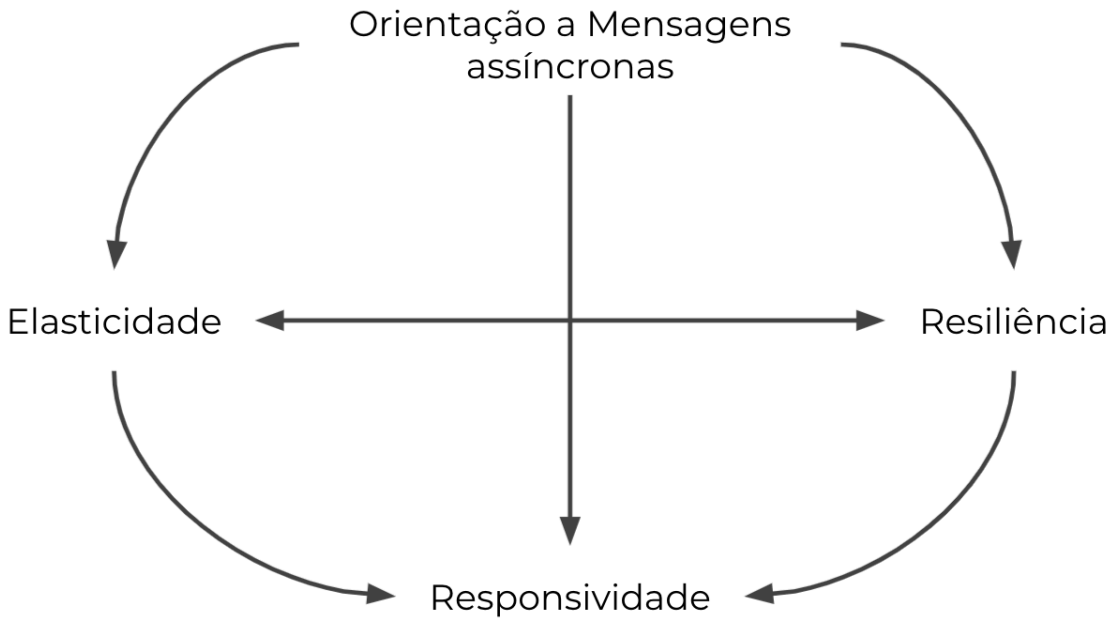

Figura 6.2: Ilustração estilizada do Manifesto Reativo.

A reatividade é um conjunto de princípios e práticas para a criação de sistemas coesos. Ela é uma especialização dos microsserviços de modo que todo sistema reativo é também um sistema de microsserviços, mas a recíproca não é verdadeira.

No coração da reatividade está a composição de grandes sistemas por pequenos serviços e das propriedades reativas de cada um deles de forma que essas propriedades se apliquem em todos os níveis de abstração e escalas, tornando-os combináveis.

6.1 Propriedades reativas

O modelo trata da construção de sistemas compostos de componentes ou serviços que são fundamentalmente Orientados a Mensagens e oferecem alta qualidade de serviço ao serem resilientes e elásticos.

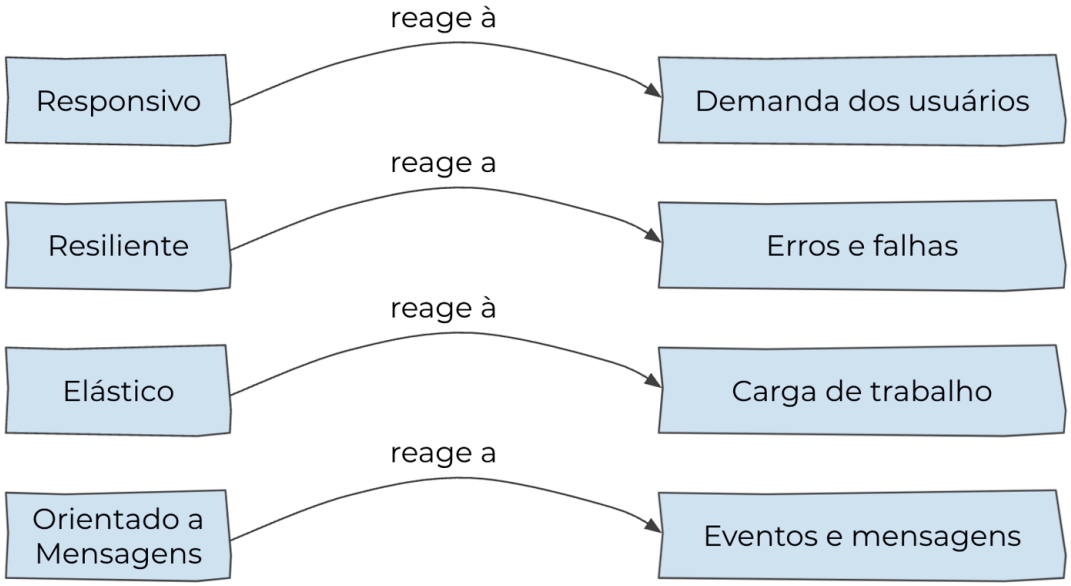

Figura 6.3: Reação das propriedades reativas.

As propriedades reativas são requisitos de qualidade de serviço. Esses requisitos devem ser capturados e implementados no projeto como quaisquer outros. Eles podem ser resumidos em elasticidade e resiliência para que o serviço seja responsivo, característica alcançável com Orientação a Mensagens assíncronas.

Essa turma do barulho deve ser indissociável e, no mundo reativo, quando falar de um, os outros devem vir à cabeça automaticamente. É como falar do John Lennon e lembrar do Paul McCartney ou falar do governo brasileiro e lembrar da corrupção.

Orientação a Mensagens

Para alcançar o Santo Graal da arquitetura reativa, não há outra alternativa, é necessário fazer uso de **mensagens assíncronas**.

Assim, estabelecem-se fronteiras bem-definidas entre os componentes, garantindo baixo acoplamento, isolamento, e transparência na localização. Mensagens assíncronas fornecem

meios para delegar o tratamento de erros através das próprias mensagens.

Os sistemas reativos devem ser Orientados a Mensagens para, explicitamente, modelar as filas do sistema e gerenciar a cadência quando necessário. Além disso, permite um melhor tratamento de demanda, escalabilidade e controle de fluxo, promovendo a elasticidade.

A comunicação deve ser assíncrona para fornecer ao cliente a opção de fazer outro trabalho em vez de ficar bloqueado esperando pela disponibilidade do recurso. Isso pode ser atingido por emissões de mensagens quando o recurso estiver disponível ou a operação for finalizada. Além de outros motivos, permite que destinatários consumam os recursos somente quando ativos, evitando desgaste do sistema.

Ao adotarmos a Orientação a Mensagens, obtemos o luxo de não nos importar com algumas dificuldades usuais. As mensagens são imutáveis e *thread-safe*, além disso, elas enfraquecem o acoplamento entre os componentes do sistema e suportam a escalabilidade.

Para sintetizar, o sistema deve permitir que você escute uma boa música e dê uma rápida pedalada enquanto aguarda a resposta do *crush* sem precisar ficar plantado com a orelha no celular.

Essa característica será melhor abordada na parte *Orientação a Mensagens*.

Elasticidade

Elasticidade é a **responsividade sob demandas extremas**. O sistema deve ser capaz de ser escalável, seja adicionando/removendo núcleos em uma única máquina ou

adicionando/removendo máquinas. O intuito é atender a demandas variáveis proporcionalmente alocando ou desalocando recursos. Tão importante quanto aumentar os recursos é diminuí-los para que o sistema permaneça sempre com o necessário. Como diriam os sonoros versos do Balu em Mogli: O Menino Lobo: "*eu uso o necessário, somente o necessário, o extraordinário é demais* 🎵".

Quando se trata de computação em nuvem, é fundamental alocar somente o necessário, ainda mais com a cotação cada vez mais exorbitante do dólar. Essa prática permite a eficiência de recursos e a economia quando os custos são pagos por demanda. Dimensionar os recursos para acomodar a demanda mínima é um tiro no próprio pé e dimensionar recursos com base no uso de pico é um tiro no joelho, no mínimo.

Sistemas precisam ser adaptativos, ou seja, sem intervenção humana e sem reconfiguração manual. Devem permitir autoescala, replicação de estado e comportamento, balanceamento de carga e atualizações.

Figura 6.4: Um ser humano elástico. Com poderes titânicos.

Veremos esse tópico mais a fundo na parte *Elasticidade*.

Resiliência

Resiliência é a **capacidade de voltar ao seu estado natural**, principalmente após alguma situação crítica e fora do comum. É ser a colher do Neo e da criança careca em Matrix, ou a colher voltando à perfeição mesmo após uma longa sessão de entretenimento com Uri Geller.

Figura 6.5: Não há colher.

Quando o assunto é a reatividade, a resiliência vai além da *responsividade sob falhas*. Não é apenas sobre uma degradação harmoniosa. Isso requer isolação e retenção da falha para evitar a contaminação dos vizinhos, o que pode originar catastróficas falhas em cascata.

A chave para alcançar sistemas autocuráveis é permitir que as falhas sejam contidas, traduzidas como mensagens, enviadas para

componentes supervisores e monitoradas por um contexto seguro externo.

Exceções não devem cruzar uma fronteira de microsserviços. Erros devem ser passados como mensagens entre componentes e tratados como conceitos de primeira classe. Se não obtivermos uma resposta de um componente, devemos suspeitar de que ele falhou e agir de acordo.

Para tal, ser Orientado a Mensagens é crucial. Impossibilita o acoplamento forte de componentes e cadeias de chamadas profundamente aninhadas. Ao remover as falhas da cadeia de chamada, libera-se o serviço cliente da responsabilidade de tratar os erros do servidor.

Veremos esse assunto mais à frente quando aprofundarmos o tema *Resiliência*

Responsividade

Os sistemas encontrados hoje devem poder ser executados por todo tipo de máquinas, potentes ou não, numerosas ou não, geograficamente próximas ou não.

Ao mesmo tempo, as expectativas dos usuários são cada vez mais difíceis de serem satisfeitas, porque o ser humano é bicho ruim e está sempre pleiteando mais. Mas não se engane, esse é o único objetivo da aplicação e da empresa para a qual você trabalha.

Fundamentalmente, não há relevância na resposta se ela não estiver disponível quando for solicitada. Os sistemas reativos almejam a responsividade acima de tudo e, mesmo sob falhas e demandas extremas, têm que rebolar para responder de forma que usuários e corporações possam depender e confiar.

Figura 6.6: Sistemas reativos, não confundir com sistemas rebolativos.

Esse tema voltará à tona mais à frente quando aprofundarmos o tema *Responsividade*.

Padrão de projeto *Observer*

Pessoalmente, percebo uma clara representação da reatividade na maneira que os sistemas trocam mensagens. Há uma inversão na perspectiva convencional de permutação de informação.

Mais uma vez recorrendo à analogia com a programação Orientada a Objeto, costumo traçar um paralelo entre a reatividade e o padrão de projeto ***Observer***, apresentado no livro *Padrões de Projeto: Soluções Reutilizáveis de Software Orientado a Objetos*, mas aplicado em um nível de abstração mais alto.

Para efeitos de comparação, copio e colo a motivação e a aplicação do padrão de projeto segundo a Wikipédia

(https://pt.wikipedia.org/wiki/Observer, acessado em 12 de abril de 2021):

Motivação: um objeto que possua agregações deve permitir que seus elementos sejam acessados sem que a sua estrutura interna seja exposta. Como garantir que objetos que dependem de outro percebam as mudanças nele? O objeto de interesse deve notificar os observadores quando for atualizado.

Os objetos devem se interligar para que só se conheçam em tempo de execução. Solucionar isso fornece uma implementação muito flexível de acoplamento de abstrações.

Aplicação: o padrão pode ser usado quando uma abstração é necessária para acessar informações em objetos separados para que variem e sejam reusados separadamente e também quando a mudança de um objeto exigir mudanças em outros, mas você não sabe quantos ou quando os informar. Em outras palavras: esse padrão pode ser usado quando se deseja um acoplamento fraco e atemporal entre os objetos.

Se trocarmos "objeto" por "serviço", esses parágrafos passariam a explicar os sistemas reativos e nem saberiam.

6.2 Conclusão

Figura 6.7: Moisés pregando sobre a reatividade.

Sistemas reativos não são novos, sendo possível rastrear relatos dessas propriedades desde as décadas de 70 e 80, pré-história computacional. Contudo, a segunda versão do Manifesto Reativo, essa que mostrei neste capítulo, nasceu apenas em 2014. A primeira versão se encontra no limbo computacional ao lado dos amantes do Windows Vista. Relíquias que nem os Indiana Jones da computação conseguiriam encontrar.

Ainda assim, apenas há poucos anos a indústria da tecnologia tem repensado sobre associá-los com as melhores práticas para desenvolvimento de aplicações distribuídas.

Ao ler sobre sistemas reativos, é comum pensar na necessidade de um sistema de fila de mensagens. Não é o caso. Qualquer mecanismo de comunicação que proporcione uma fronteira assíncrona é suficiente. Pode ser um produto de mensagens, mas também podem ser *websockets* síncronos, ou um protocolo assíncrono utilizando ferramentas convencionais.

Certa vez, Martin Fowler – ou Tintin, como eu gosto de chamá-lo sem sua autorização – disse algo interessante ao falar sobre distribuição (https://martinfowler.com/bliki/FirstLaw.html):

> *Primeira lei do design de objetos distribuídos: "não distribua seus objetos"*. - Martin Fowler

Acredito que possamos dizer justamente o contrário no contexto da reatividade. Como o Manifesto Reativo atesta, sistemas reativos não são uma questão de escolha, mas de necessidade.

Ao passo que a reatividade é uma abordagem que não é possível ser aplicada em todos os sistemas, uma vez que depende de operações que fogem ao seu controle, eu diria, com o poder investido em mim, perante mim mesmo:

> Primeira lei dos sistemas reativos: "quando possível, vá fundo.".

Capítulo 7
Dificuldades da reatividade

Dói admitir, mas nem só de flores vive a reatividade, camaradas.

Costumo comparar responsividade com os produtos Activia, deliciosos, caros e, se não usados corretamente, capazes de gerar efeitos catastróficos. Por se basearem nos microsserviços, os sistemas reativos são automaticamente complexos e, por conta das suas peculiaridades, podem ser tão frustrantes quanto repor o estoque de papel higiênico após o incidente do Activia.

Alguns pontos que costumam ser mencionados como dificuldades habituais provenientes da arquitetura reativa são a **distribuição de dados**, o **desempenho** e a **conversão à reatividade**. Quanto à dificuldade relacionada ao uso do Activia, prefiro não comentar.

Figura 7.1: Conversão à reatividade.

7.1 Distribuição de dados

Ao escrever aplicativos monolíticos, a consulta em um único banco de dados para todos os fins é trivial. Muitos benefícios dos microsserviços, no entanto, dependem de serem desenvolvidos e funcionarem autonomamente, com uma arquitetura descentralizada, seguindo o princípio do Repositório Único. Como veremos nos

próximos capítulos, ao optar por uma arquitetura descentralizada, descentralizam-se também os dados de persistência. Logo, uma estratégia de gerência de dados deve ser traçada para o acesso aos dados distribuídos.

Os sistemas reativos, reforçando o aspecto da resiliência, sugerem a *duplicação* de dados para antecipar uma busca futura e evitar desastres. Mas, apesar de os termos *duplicação* e *replicação* serem bastante utilizados, não julgo serem os mais adequados para definir a técnica. *Reinterpretação persistida* e *pré-processamento*, penso, sob o risco de ser cancelado pela internet, são expressões preferíveis. Não são apenas um *ctrl+c* e *ctrl+v*. Os dados, oriundos de outro domínio, são lapidados e persistidos no domínio do serviço que recebe a mensagem. Estão, portanto, sujeitos a sua ótica e reinterpretação.

Por exemplo, enquanto em um *e-commerce* o `serviço-de-produtos` sabe que um café custa somente três reais, esse mesmo dado, visto sob a perspectiva do `serviço-de-preços`, informa que três reais é o custo de um café. A despeito de representarem quase a mesma informação, cada serviço interpreta da maneira que lhe é mais conveniente.

A técnica possibilita que, ao receber mensagens do `serviço-de-preços`, o `serviço-de-produtos` identifique que tanto o café quanto o cappuccino custam três reais e, por isso, organize os dados de acordo com a estrutura mais apropriada para o contexto, priorizando desempenho na busca, por exemplo. O processamento prévio antes que essa informação seja requisitada permite tais otimizações sem impactar a requisição.

Seja *reinterpretação*, seja *replicação*, a técnica pode soar contraintuitiva e desperdício espacial, indo contra o que colocamos em prática nas normalizações de bancos de dados. Mas não é. Essa é uma estratégia válida e não apenas nos sistemas reativos.

Desnormalização dos dados

Discutir desnormalização passa pela compreensão de um modelo de dados normalizado. O termo "normalizar" se refere ao processo de organizar os dados de forma que cada uma de suas entidades apresente um único significado e esteja livre de redundâncias, maximizando a acessibilidade.

Para atingir esse objetivo, chamado de **normalização**, usamos um processo composto por um conjunto de regras distintas e sucessivas, aplicadas sobre cada atributo das entidades, conhecidas como *formas normais*.

Como o nome sugere, a **desnormalização** é o oposto da normalização e significa colocar deliberadamente os mesmos dados em vários lugares, aumentando a redundância. Apesar do desperdício espacial por conta do pleonasmo de informações, as aplicações são beneficiadas com o ganho de desempenho por não ser mais necessário requisitar informação de outro lugar e são capazes de pré-processar dados, o que reforça pontos benéficos dos sistemas reativos, como a resiliência e a autonomia.

Um modelo de dados desnormalizado não é o mesmo que um modelo de dados que não foi normalizado. A desnormalização só deve ocorrer após um nível satisfatório de normalização.

Duplicar alguns dados em vários microsserviços não é uma decisão incorreta. Ao contrário, ao fazer isso, você pode converter as informações nos termos específicos do domínio, lapidando arestas desnecessárias e moldando ao contexto necessário.

A desnormalização não é uma pílula mágica. Apesar de tentadora e eficiente na teoria, ela também traz uma série de desvantagens e anomalias inerente que precisam ser controladas:

- **Espaço de armazenamento extra:** mais espaço será utilizado para a persistência do valor. Mesmo baixos, os custos de armazenamento devem ser comparados ao preço da indisponibilidade e do mau desempenho.

- **Documentação adicional:** manter dados reinterpretados em mais de um local requer um mapeamento de onde eles estão.
- **Operações mais lentas:** os pré-processamentos e cálculos antecipados aceleram as recuperações de dados, mas, ao mesmo tempo, retardam as atualizações assíncronas. Por se tratar de tarefas assíncronas, o tempo adicional é relativizado.
- **Inconsistência:** desnormalizar pode nos levar a inconsistências, como dados errados e até contraditórios. Distribuir os dados pode implicar em mantê-los atualizados. Conforme são distribuídos e se amontoam, é mais fácil perder o controle sobre o que e quando deve ser atualizado e se a informação persistida é ou não obsoleta. Na parte *Elasticidade*, explicarei como essa desvantagem pode afetar os microsserviços e como mitigá-la.

Figura 7.2: Tarefas hercúleas.

7.2 Desempenho reativo

Outra dificuldade alegada decorrente de uma estrutura baseada nos microsserviços é o desempenho. Devido à distribuição de dados e à adição da camada de rede, operações que costumavam ser quase instantâneas são afetadas pela falta de garantias da internet e pelo tempo de resposta dos outros serviços.

O fato de empregar o conjunto de regras para tornar o ecossistema reativo, por outro lado, não interfere no desempenho geral do produto.

Existe um mito, no entanto, que insiste em dizer o contrário. O mito afirma que o emprego de mensagens assíncronas introduz um retardo no tempo das transações. Sob o prisma da lógica, é o mesmo que afirmar que o Gugu Liberato teve envolvimento com o assalto ao Banco Central do Brasil em Fortaleza. Ambas são afirmações que não encontram argumentos razoáveis que as sustente.

O fato de ser assíncrono não tem relação direta com desempenho ou dedicação do processamento de solicitação. Existem muitos *trade offs* entre as abordagens assíncronas e síncronas, mas desempenho, no fim das contas, não é um deles.

A abordagem síncrona significa que o tempo limite é implementado em uma camada inferior, ou seja, você não pode continuar executando enquanto espera pela sua resposta.

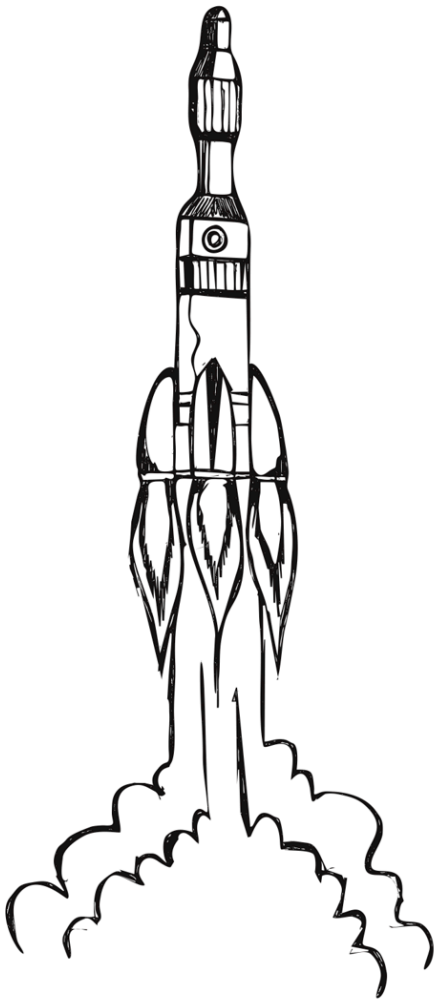

Figura 7.3: Arriba, abajo, al centro, y a la luna!

A assincronia requer a implementação do tempo limite na camada de lógica de negócios. Tecnicamente, é possível esperar por uma resposta da mensagem assíncrona e, às vezes, é necessário. Existem momentos em que, para suprir uma demanda síncrona, precisamos de um agente que bloqueie o fluxo até ter uma resposta, sendo ela de expiração ou não. Fundamentalmente, isso faria dela uma mensagem síncrona. Abordaremos mais esse assunto quando tratarmos sobre as condições de corrida nos microsserviços, na última parte do livro.

Nas chamadas síncronas, quando há uma cadeia de mensagens, a solicitação original não obtém uma resposta até que todas as internas sejam concluídas. Considerando um aumento do número de chamadas e um atraso em uma chamada intermediária, concluímos que o desempenho terá um prejuízo direto.

Em um sistema de microsserviços repleto de cadeias síncronas, quanto mais numerosas e longas forem as cadeias, menos autônomo e mais sujeito a intempéries será o desempenho do sistema. O risco dessas pequenas agentes do anticristo é multiplicado e somado ao custo das transmissões das mensagens, ocasionando uma alta probabilidade de perda de performance.

Portanto, se alguma associação pode ser estabelecida, é a oposta: regimes síncronos implicam em maiores chances de piorar o desempenho.

Nos sistemas reativos, por conta do desacoplamento temporal com mensagens assíncronas, dado que os serviços só consomem as mensagens disponíveis quando estiverem desimpedidos e que os dados necessários são imediatamente acessíveis, a abordagem pode impedir gargalos de processamento, evitando lentidão devido à sobrecarga por estresse dos componentes internos.

O desempenho e a distribuição de dados, apesar de complexos, não são derivados dos sistemas reativos. São frutos da adoção de uma escolha prévia, a filosofia dos microsserviços. Em contrapartida, os sistemas reativos servem de orientação e podem amenizar esses problemas. Eles providenciam uma maneira extremamente flexível de criar aplicativos, visando sempre à responsividade.

Associar falta de desempenho e o desperdício espacial à reatividade não passa de uma daquelas histórias para boi dormir enraizada na cultura de desenvolvimento.

Outros exemplos de falácias na cultura de desenvolvimento é afirmar que as soluções-alternativas-improvisadas-de-emergência são temporárias e serão removidas em breve ou que o código só tem

mais um bug para ser corrigido e depois ficará perfeito. Sabemos que os deuses do desenvolvimento não gostam de ser desafiados e é questão de tempo até essas afirmações se tornarem falsas.

A negação dessa falácia, contudo, não implica na afirmação do contrário. A reatividade, por si só, não causa melhor desempenho ou aproveitamento espacial.

7.3 Conversão à reatividade

Em um sistema recém-nascido, aplicar os ensinamentos do mundo reativo é simples e intuitivo. Basta não reproduzir o comportamento do Burro do Shrek e não criar interações proativas onde possível. Converter uma arquitetura proativa madura para uma reativa, em compensação, faz soar as trombetas do apocalipse e nos fazer querer fugir para bem longe. Sei lá, talvez para o México?

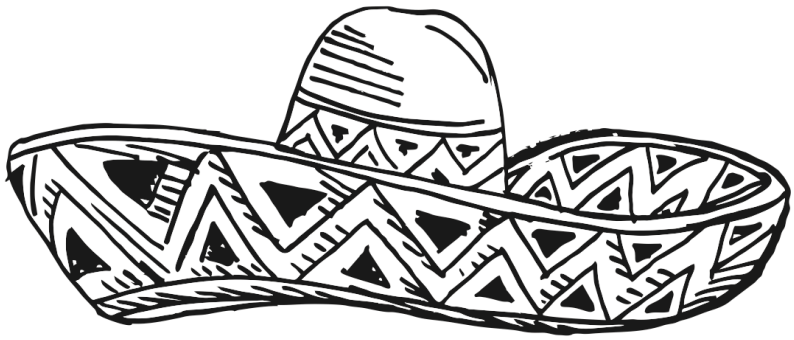

Figura 7.4: Ai, caramba!

O que se pode fazer, então? Eu gostaria de poder dizer que há uma resposta fácil. Que é só ligar os pontinhos, apertar ⬆⬆⬇⬇⬅ ➡ ⬅ ➡ e gritar "*arriba muchacho*". Mas a realidade não costuma ser tão generosa na computação. A prova disso é que até hoje a realidade

computacional nos humilha com as atualizações automáticas do Windows em momentos inapropriados.

Algoritmo para reatividade

Calma, calma, não criemos pânico. Felizmente, há um algoritmo que se pode executar para amenizar o problema da conversão à reatividade. Quanto às atualizações do Windows, resta se entregar à embriaguez com uma boa tequila.

A estratégia se baseia na técnica de **estrangulamento de sistemas**. Ela consiste em, paulatinamente, reinterpretar e copiar localmente os dados de um sistema para partes específicas de outro. Conforme os recursos são replicados, a obrigação de consultar dados em outro sistema é reduzida, eventualmente, não sendo mais sequer necessária.

1. Fazendo uso, de novo, do princípio *Tell, don't ask*, identifica-se uma transação **proativa** que solicite informação. Por motivos didáticos, nomearemos como `serviço-solicitante` o serviço que atua proativamente pedindo a informação; e `serviço-detentor`, o serviço que detém a informação e a repassa para o `serviço-solicitante`.

2. Verificar como o dado da transação é utilizado pelo `serviço-solicitante`, se pode ser de maneira reativa. Se não puder, retorne ao passo 1.

3. Replicar os valores requisitados da base de dados do `serviço-detentor` para o banco de dados do `serviço-solicitante`, aproximando o `serviço-solicitante` dos dados de que ele precisa.

4. Fazer com que o `serviço-solicitante` passe a observar e **reagir** a qualquer alteração realizada nesses dados no `serviço-detentor`. Dessa forma, na próxima vez em que o `serviço-solicitante` precisar dessas informações, haverá a possibilidade de já as possuir sem requisitar novamente.

5. O `serviço-detentor` deve informar sempre que houver alterações relevantes sobre esses valores.
6. Retornar ao passo 1 enquanto houver transações passíveis de conversão.

Como diz Alex Zanetti, um amigo da firma: "no PowerPoint, tudo compila". O que quer dizer que, na teoria, isso é bem simples, mas, surpreendendo o total de zero pessoa, na realidade o buraco é mais embaixo.

A replicação é como a dança do créu e a lacraia, sabe? Tem várias velocidades. A velocidade escolhida para a replicação de dados (passo 3) é de suma importância.

A depender de aspectos, como o tipo dos bancos de dados envolvidos, importância dos dados, complexidade, quantidade de valores e paciência da chefe, a migração dos dados de uma base para outra pode ter um ritmo aceleradíssimo, tipo *Through the fire and flames* do *Guitar hero 3*, ou um mais lento, quase parando, como qualquer coisa do Chico Buarque.

Por isso, a cadência requer atenção e possui todo um espectro a ser explorado. Uma escolha malfeita pode ter como consequências migrações de dados que duram uma eternidade (o que pode não ser um problema), frituras de banco de dados e degradação do desempenho das demais transações do sistema.

Nos dois extremos do espectro, temos:

- **Ritmo acelerado possuído pelo espírito ragatanga:** recomendado quando a urgência não permite outra escolha ou quando são poucos dados envolvidos. Geralmente, um *script*, ou algo que execute o papel de um, copia os valores originais, reinterpreta e os persiste no banco de dados do `serviço-solicitante`. Ao final, altera-se o comportamento do `serviço-solicitante` para que ele busque a informação no seu banco de dados em vez de solicitar ao `serviço-detentor`.

- **Ritmo devagar, *despacito*:** altera-se o `serviço-solicitante` para verificar se há uma cópia local da informação desejada antes de qualquer solicitação ao `serviço-detentor`. Caso não haja, segue o fluxo anterior para solicitá-la e, uma vez que a informação está sob posse do `serviço-solicitante`, ele a reinterpreta e a persiste no seu banco de dados local. Na próxima vez em que esse dado for necessário, o `serviço-solicitante` não precisará solicitá-la ao `serviço-detentor`. Esse ritmo permite um desmame mais suave e menos arriscado.

  Idealmente, adicionam-se métricas para saber quando a consulta proativa deixa de ser necessária ou atinge níveis aceitáveis. Somente nesse momento será possível remover a requisição por elas. É migração sob demanda, onde os dados usados que são realmente usados são migrados quando necessários.

  Migrações superlentas são mais comuns do que se pode imaginar, podem durar meses e até anos. Dessa forma, é necessário aprender a conviver com sistemas sempre em migração e dados incompletos e inconsistentes espalhados por vários microsserviços. Nesses cenários caóticos, é fácil se perder em meio a tanta informação pulverizada. Para facilitar a administração, uma boa observabilidade é mandatória, como veremos no capítulo *Observabilidade de sistemas*.

Os dois extremos são pontos válidos do espectro. É muito comum também a adoção de um meio-termo. Uma heurística se faz necessária para determinar as prioridades na migração: primeiro os dados mais frequentemente utilizados? Ou os mais utilizados nos últimos meses? Quem sabe, o contrário: primeiro os menos utilizados, aproveitando para fazer a validação da técnica em dados que, em teoria, têm menos valor.

Qualquer que seja o ponto escolhido do espectro, é necessário fazer com que, como os sistemas reativos exigem, a informação mantenha-se atualizada onde quer que ela esteja replicada. Ou seja,

o `serviço-detentor` deve informar sempre que houver alterações relevantes nos dados e o `serviço-solicitante` deve estar sempre atento.

7.4 Conclusão

Os problemas expostos representam boa parte dos argumentos utilizados contra a adoção da reatividade. Embora tenham alguma fundamentação, eles estão relacionados a uma escolha prévia: a utilização da filosofia de microsserviços. A reatividade em si não corrobora com os problemas mencionados.

No entanto, existem sim, infelizmente, outras dificuldades intrínsecas à adoção da reatividade que podem ser tão ou mais difíceis do que os pontos mencionados. Não é fácil, por exemplo, tratar a **consistência dos dados**. O que fazer quando a informação no `serviço-solicitante` não está atualizada conforme o valor no `serviço-detentor` ?

Além disso, há outros problemas comuns, semelhantes aos *deadlocks*, *loops* infinitos e condições de corrida. A esses pontos, como são vastos e exigem mais profundidade, dedico inteiramente as próximas partes do livro.

A boa notícia é que dificuldades bem maiores, como o sentido da vida, do universo e tudo mais, ou a teoria da relatividade não têm nada a ver com sistemas reativos.

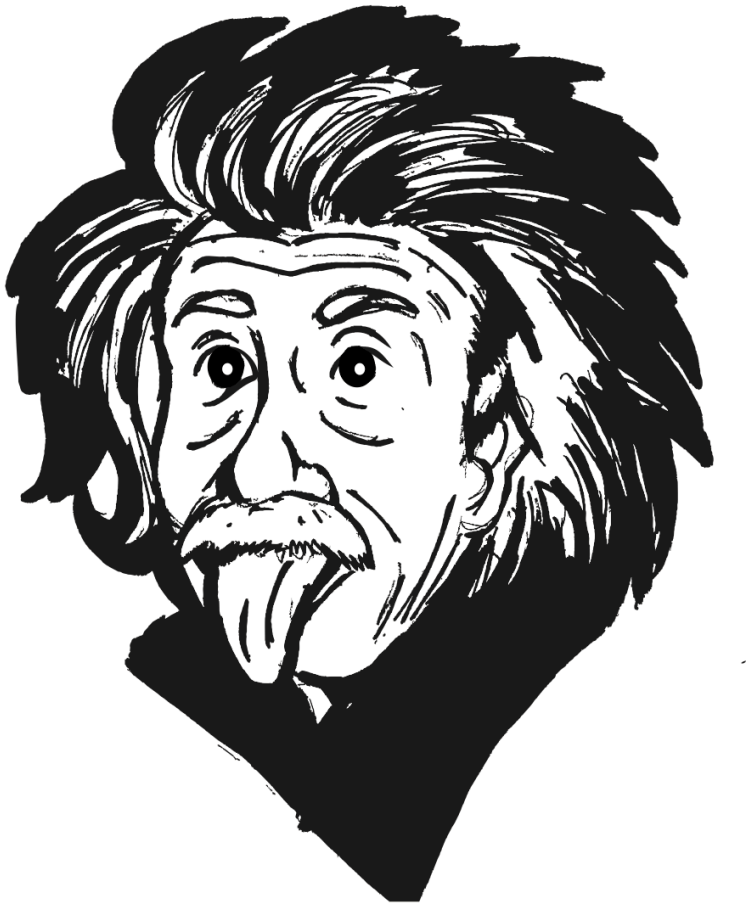

Figura 7.5: Sistemas reativos, não confundir com sistemas relativos.

Capítulo 8
Orientação a Objeto Orientada a Mensagens Reativas

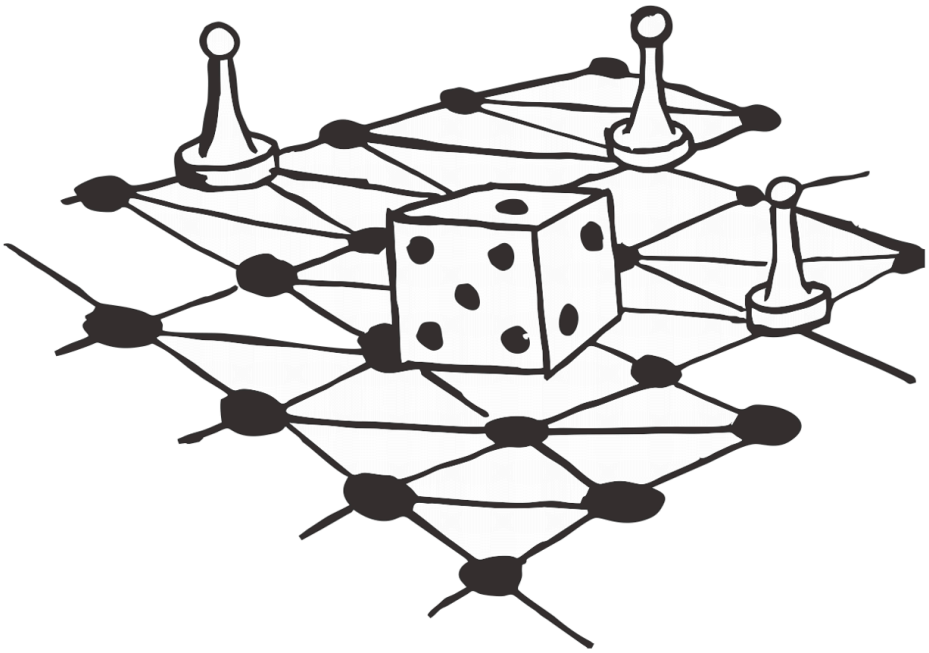

Figura 8.1: Jogo de tabuleiro.

Vimos anteriormente um estudo que traça um paralelo entre a aplicação dos princípios de OO e a filosofia de microsserviços. No capítulo anterior, discutimos sobre algumas das ditas dificuldades encontradas ao se desenhar projetos reativos, como desempenho, consistência e distribuição de dados.

Este capítulo, além de ajudar a desmascarar as falácias mencionadas no capítulo anterior e de ter um propósito terapêutico e lúdico, nasceu para saciar uma curiosidade pessoal. Como seria desenvolver um projeto OO usando práticas dos sistemas reativos, sobretudo a Orientação a Mensagens assíncronas?

Sabendo da relação entre arquitetura de microsserviços e OO, o que aprenderíamos ao simular um ecossistema reativo no nível de abstração de Orientação a Objeto? Quais seriam os desafios e aprendizados encontrados?

Seria um projeto bem didático, uma vez que o design de OO está mais próximo do nosso cotidiano e muito palpável, portanto fácil de assimilar. Na simulação, o importante seria o código, logo sem infraestrutura, sem *pipeline* ou *cloud providers* reais, pontos indispensáveis ao universo dos microsserviços.

Nesse nível de abstração, as classes funcionariam como serviços. Os objetos seriam as instâncias de cada serviço, e o gerenciador de mensagens (*mediator*, *broker*, *delegate*, chame como quiser) simularia um protocolo de comunicação assíncrona que exigiria qualquer intermediário (*amqp*, *kafka*, *sns/sqs*, *mqtt* etc.).

Carinhosamente, batizei esse paradigma de **Orientação a Objeto Orientada a Mensagens Reativas™**, ou simplesmente OOOMR, para os íntimos.

Por sua natureza OO, o projeto permite que o próprio código – finalmente um pouco de código para saciar seu vício, leitor, leitora – seja compartilhado para representar a arquitetura reativa. Em vez de diagramas para os microsserviços, teremos classes e OO.

Como o nome induz, a OOOMR possui muitas semelhanças com o recém-estabelecido paradigma da Programação Reativa. Como veremos mais à frente, assim como a Programação Reativa, a OOOMR também promove o acoplamento fraco e atemporal. Porém, enquanto a programação reativa prega a reatividade pelo uso de *data streams* e componentes observadores e observáveis, a OOOMR não passa de uma adaptação emergencial da OO feita com o intuito lúdico e didático de traçar uma analogia entre a construção de sistemas OO e os sistemas reativos.

8.1 Regras da OOOMR

Figura 8.2: Sistemas reativos, não confundir com sistemas regulativos.

Para a implementação da OOOMR, elaborei três regras:

1. **Abolir todos os *getters*:** o método `get` simboliza a própria reencarnação da proatividade em forma de Orientação a Objeto. Nessa abordagem, é o demônio em forma de código.

   O método funciona literalmente como um meio para solicitar proativamente um dado que pertence a outro objeto, contrariando o princípio *Tell, don't ask* e indo frontalmente

contra o conceito de reatividade. Evitar `get` na OOOMR, evita também a repetição do comportamento do Burro do Shrek: "A gente já chegou?".

A adoção dessa regra automaticamente nos transforma em, como fala o recorrentemente citado Martin Fowler, erradicadores de *getters* (leia mais sobre o assunto em https://martinfowler.com/bliki/GetterEradicator.html).

2. **A comunicação entre objetos de negócio deve ser feita exclusivamente por mensagens**, isto é, sem métodos públicos. Classes de negócio, excluindo as *DTO*, *VO*, *POJO* – (insira outra da mesma categoria aqui) – que simulam apenas as mensagens serializáveis, não sabem da existência umas das outras, excetuando-se, obviamente, a classe controladora de mensagens. Essa regra permite que o sistema seja executado de maneira assíncrona, desacoplada e independente.

   Analisando enquanto escrevo, percebo que essa regra eliminaria a necessidade da primeira. Como quero enfatizar meu reiterado asco aos *getters*, resolvi deixar a primeira regra lá bonitinha.

3. **Mensagens devem ser *stateless*:** se você implementa sistemas com APIs *RESTful*, essa vai ser tranquila. Isso significa que as mensagens devem incluir todas as informações suficientes e necessárias para que os receptores as processem de maneira adequada.

Definidas as regras que eu deveria seguir para a implementação do projeto, é hora de sentar a bota.

Hora de sentar a bota

Devido a uma antiga paixão, o desenvolvimento de jogos, o projeto se trata de uma réplica do mundialmente conhecido **Tetris** (https://tetris.com).

Para os que foram abduzidos ou, por algum outro motivo desconhecido, estiveram ausentes do plano terreno nas últimas

décadas, Tetris foi e é um jogo eletrônico muito popular, ancestral remoto do Minecraft, febre entre os *gamers* das gerações anteriores, um dos jogos mais influentes de todos os tempos. O jogo consiste em um tabuleiro de 10 células de largura por 20 células de altura, cujo objetivo é amontoar os tetraminós (nome dado às peças de 4 células no jogo), que descem pelo tabuleiro. Quando uma linha horizontal está totalmente repleta de células, ela é eliminada, todas as células acima da linha eliminada descem um nível, e o jogador ganha pontos. Quando a pilha de peças chega ao topo do tabuleiro, a partida se encerra. Simples e divertido. Aqui vai uma curiosidade para vocês, o que prova que este livro também tem cultura: dá-se o nome "tetris" ao movimento que elimina quatro linhas de uma vez só.

O projeto é executável nos *browsers*, assim o jogo é multiplataforma de brinde. Foram utilizados uma linguagem de programação compatível e orientada a objeto, TypeScript (https://www.typescriptlang.org/), e a biblioteca de jogos *phaser* (https://phaser.io/).

A seguir, serão apresentados trechos de código. São resumos das implementações reais, porque decidi remover o que prejudicava a didática. Caso queiram se divertir com o código ou analisar o jogo, a seguir estão os links do repositório e do jogo. Link do repositório: https://github.com/virgs/tetris. Link do jogo: https://virgs.github.io/tetris.

Ao jogarem, não permitam que meu inegável talento artístico os impressione. Apesar de tentador, lembrem-se de que o propósito do jogo era fazer um estudo exploratório.

8.2 Serviços | Objetos

Dentre os domínios criados, alguns se destacam, portanto classes correspondentes foram criadas. Elas atuariam como microsserviços no outro nível de abstração:

1. **FallingTetramino:** manuseia e verifica a colisão do tetraminó ativo, aquele que está caindo. Ao receber um comando do usuário, deve aplicá-lo à peça e movê-la pelo tabuleiro.
2. **TetraminoStack:** empilha as células inativas.
3. **TetraminoFactory:** cria o próximo tetraminó e o exibe enquanto o atual desce pelo tabuleiro.

GameController

Além dessas, é necessário criar a classe que atuaria como um orquestrador de contêiner. Ela executa os serviços/cria os objetos, administra a cadência do jogo e detecta o fim da partida.

Nessa classe, o método `create` inicializa os objetos listados anteriormente, e eles não são atribuídos a nenhum membro da instância. Na classe, não existe nenhuma referência a eles, garantindo que nenhum método deles será explicitamente chamado diretamente, apenas via mensagens. Ainda no método, após a criação de todos os componentes do jogo, a classe dá o pontapé inicial na partida emitindo a mensagem `CREATE_TETRAMINO` para comandar a geração de um tetraminó.

No método `update`, destaco os eventos `ONE_STEP_DOWN_TIME_ELAPSED` e `UPDATE_TIME_ELAPSED`, que controlam o ritmo em que as peças caem e em que os comandos do usuário são tratados, respectivamente. Eles interagem com o gerenciador de mensagens `MessageManager`, que é equivalente ao *broker* na arquitetura de microsserviços.

Ao perceber que é emitida a mensagem `TETRAMINO_STACK_NEW_CONFIGURATION` após a detecção do empilhamento de um tetraminó, como veremos, a classe verifica se alguma célula ultrapassou o teto do tabuleiro. Em caso positivo, ela encerra a partida, caso contrário, cria um tetraminó e dá continuidade ao jogo.

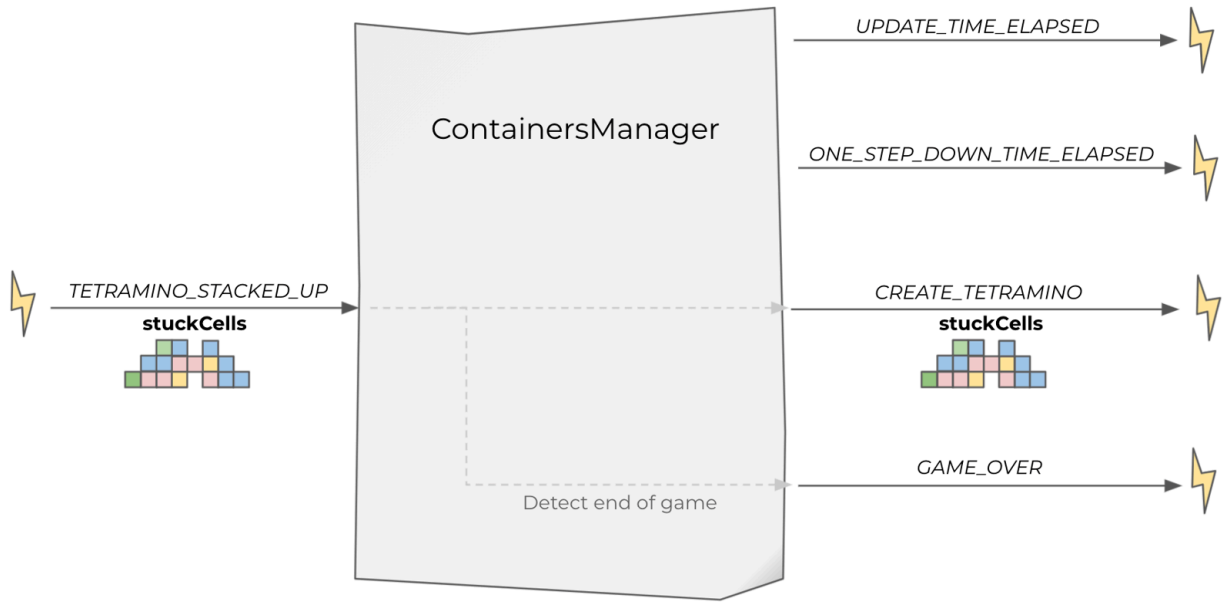

Figura 8.3: Mensagens escutadas e emitidas por GameController.

```
export class GameController extends Phaser.Scene {
  public async create(): Promise<void> {
    this.registerToEvents();
    new TetraminoFactory({scene: this});
    new TetraminoStack({scene: this});
    new FallingTetramino({scene: this});
    new InputManager({scene: this});
    MessageManager.emit(Messages.CREATE_TETRAMINO);
  }

  public update(time: number, delta: number): void {
    this.updateTimeCounterMs += delta;
    if (this.accumulatedTimeIsGreaterThanOneStepDownTime()) {
      MessageManager.emit(ONE_STEP_DOWN_TIME_ELAPSED);
    }
    MessageManager.emit(UPDATE_TIME_ELAPSED, delta);
  }

  private registerToEvents() {
    //...
    MessageManager.on(Messages.TETRAMINO_STACK_NEW_CONFIGURATION, event =>
```

```
{
      const isSomeCellOverTheRoof = event.stuckCells.some(cell =>
cell.block.y < 0);
      if (isSomeCellOverTheRoof) {
        MessageManager.emit(Messages.GAME_OVER);
      } else {
        MessageManager.emit(Messages.CREATE_TETRAMINO, {stuckCells:
event.stuckCells});
      }
    });
  }
}
```

TetraminoFactory

A próxima classe/serviço que apresento é `TetraminoFactory`, responsável por criar os tetraminós e exibir a próxima peça a entrar no jogo. Logo ao ser inicializada, a classe gera uma peça aleatoriamente e a exibe na vitrine.

Ao escutar o comando `CREATE_TETRAMINO` com a informação da configuração atual da pilha de células, a classe envia a peça da vitrine para o jogo, cria uma peça e a apresenta na vitrine.

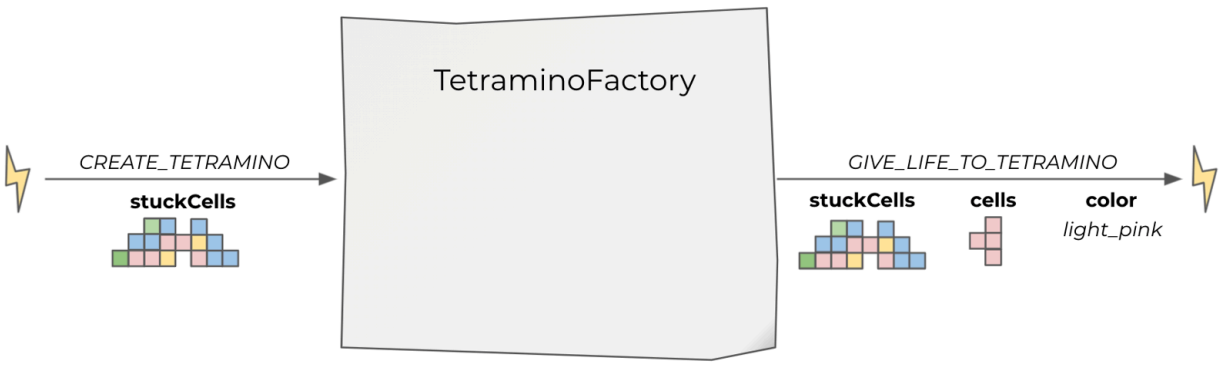

Figura 8.4: Mensagens escutadas e emitidas por TetraminoFactory.

```
export class TetraminoFactory {
  public constructor() {
```

```
    this.nextPiece = TetraminoFactory.randomlyCreateNextBlock();
    this.previewNextPiece();
    MessageManager.on(Messages.CREATE_TETRAMINO, (options: { stuckCells:
StuckCell[] }) => {
      const next = this.nextPiece;
      this.nextPiece = TetraminoFactory.randomlyCreateNextBlock();
      this.previewNextPiece();

      MessageManager.emit(Messages.GIVE_LIFE_TO_TETRAMINO, {
        cells: next.blocks,
        color: next.color,
        stuckCells: options ? options.stuckCells : []
      });
    });
  }
}
```

FallingTetramino

O objeto/instância que manuseia a peça controlável, `FallingTetramino`, escuta vários eventos e comandos:

- **UPDATE_TIME_ELAPSED:** limita a quantidade de movimentos que a peça tem por segundo. É quando os comandos do jogador são aplicados.
- **INPUT_DETECTED:** mensagem que o jogador lançou para manipular o tetraminó (cima, esquerda, rotação).
- **ONE_STEP_DOWN_TIME_ELAPSED:** a peça deve descer um nível no tabuleiro. O objeto/instância simula sua próxima posição e, caso uma colisão com alguma peça empilhada ou com o fundo seja detectada, o evento `FALLING_TETRAMINO_LANDED` é disparado com a informação da posição em que o tetraminó atual "repousa sob a pilha de células" e a sua cor.
- **GIVE_LIFE_TO_TETRAMINO:** é o comando que adiciona o tetraminó recém-criado ao jogo para que ele possa ser manuseado. O comando também informa a posição das células

empilhadas para que a detecção da colisão possa ser calculada futuramente.

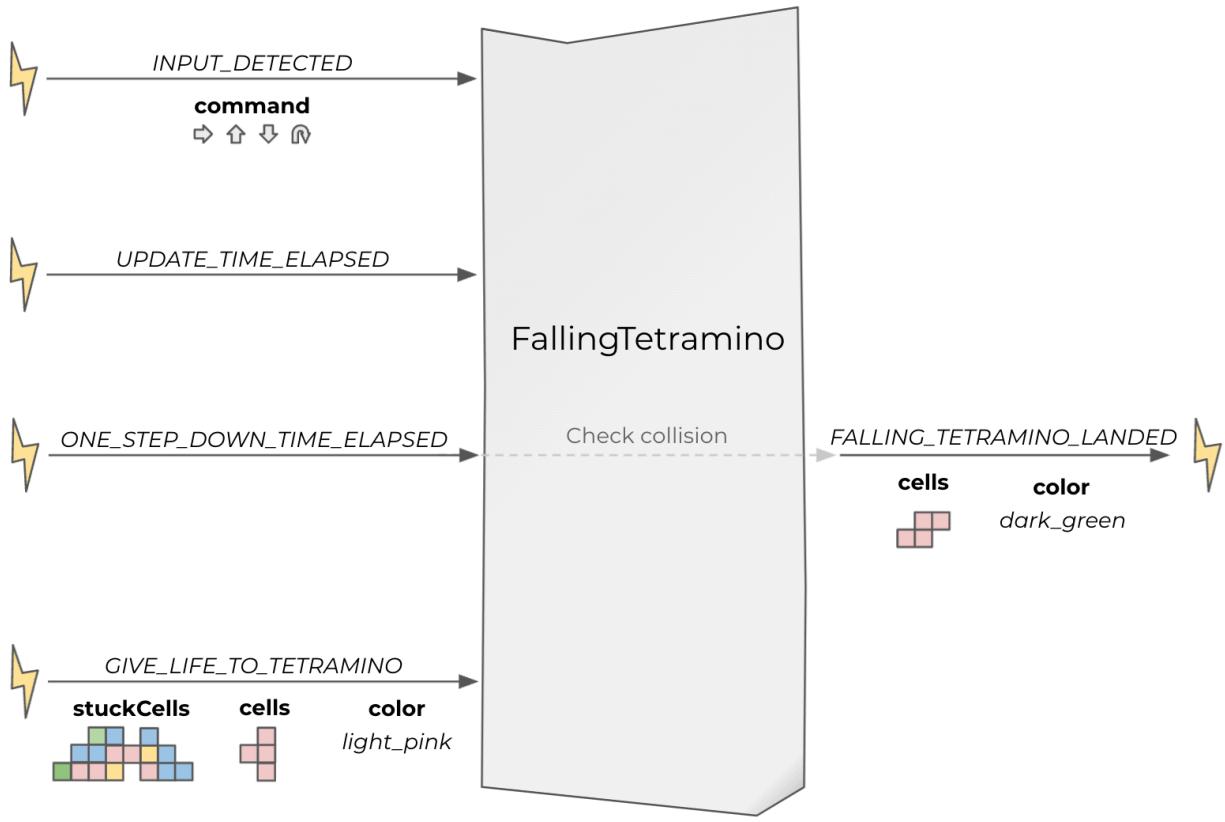

Figura 8.5: Mensagens escutadas e emitidas por FallingTetramino.

```
export class FallingTetramino {
   public constructor(options: { scene: Phaser.Scene }) {
      //...
      MessageManager.on(UPDATE_TIME_ELAPSED, () => this.update());
      MessageManager.on(INPUT_DETECTED, command =>
this.nextCommands.push(command));
      MessageManager.on(ONE_STEP_DOWN_TIME_ELAPSED, () =>
this.goDownOneLevel());
      MessageManager.on(Messages.GIVE_LIFE_TO_TETRAMINO, (event: {
         cells: Point[],
         stuckCells: StuckCell[],
         color: string
      }) => {
```

```
            this.color = event.color;
            this.nextCommand = [];
            const centerPoint = new Point(dimension.x / 2 - 1, -
(event.cells.length + 1));
            this.position = centerPoint;
            this.cells = event.cells;
            this.stuckCells = event.stuckCells.map(cell => cell.block);
         });
      }

      private goDownOneLevel() {
         const nextPosition: Point = new Point(this.position.x,
this.position.y + 1);
         if (!this.detectCollision(nextPosition, this.cells)) {
            this.position = nextPosition;
         } else {
            MessageManager.emit(FALLING_TETRAMINO_LANDED, {
               color: this.color,
               cells: this.cells
                        .map(cell => new Point(this.position.x + cell.x,
this.position.y + cell.y))
            });
         }
      }
}
```

TetraminoStack

Por fim, a classe/serviço `TetraminoStack`, responsável por gerenciar a pilha de células no tabuleiro, escuta apenas uma mensagem: `FALLING_TETRAMINO_LANDED`, o evento que relata que uma peça acabou de ser empilhada e a sua cor. Ao receber essa notificação, a classe verifica se há alguma linha completa. Se houver, você bem lembra, a entidade remove a linha e emite o evento `LINES_DELETED` informando a quantidade de linhas eliminadas.

Após a verificação de remoção, a classe emite uma mensagem `TETRAMINO_STACK_NEW_CONFIGURATION` informando a nova configuração da pilha de células.

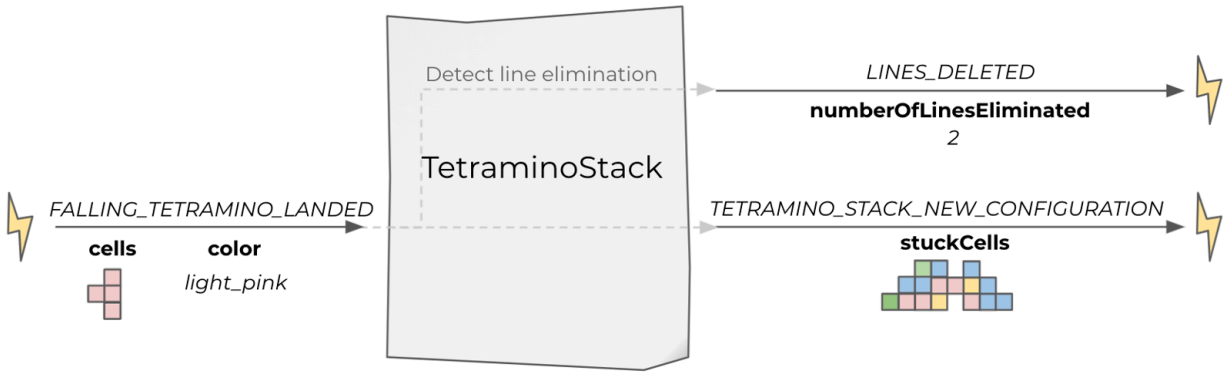

Figura 8.6: Mensagens escutadas e emitidas por TetraminoStack.

```
export class TetraminoStack {
  private registerToEvents() {
    MessageManager.on(Messages.FALLING_TETRAMINO_LANDED, (event: { cells:
Point[], color: string }) => {
      this.stuckCells = this.stuckCells.concat(event.cells
            .map(deadCell => ({block: deadCell, color: event.color})));
      const numberOfEliminations = this.detectLineElimination();
      if (numberOfEliminations) {
        MessageManager.emit(Messages.LINES_DELETED,
{numberOfEliminations});
      }
      MessageManager.emit(Messages.TETRAMINO_STACK_NEW_CONFIGURATION,
{stuckCells: this.stuckCells});
    });
  }
}
```

É interessante reparar no ciclo de mensagens entre esses serviços com:

- GIVE_LIFE_TO_TETRAMINO
- FALLING_TETRAMINO_LANDED
- TETRAMINO_STACK_NEW_CONFIGURATION
- CREATE_TETRAMINO

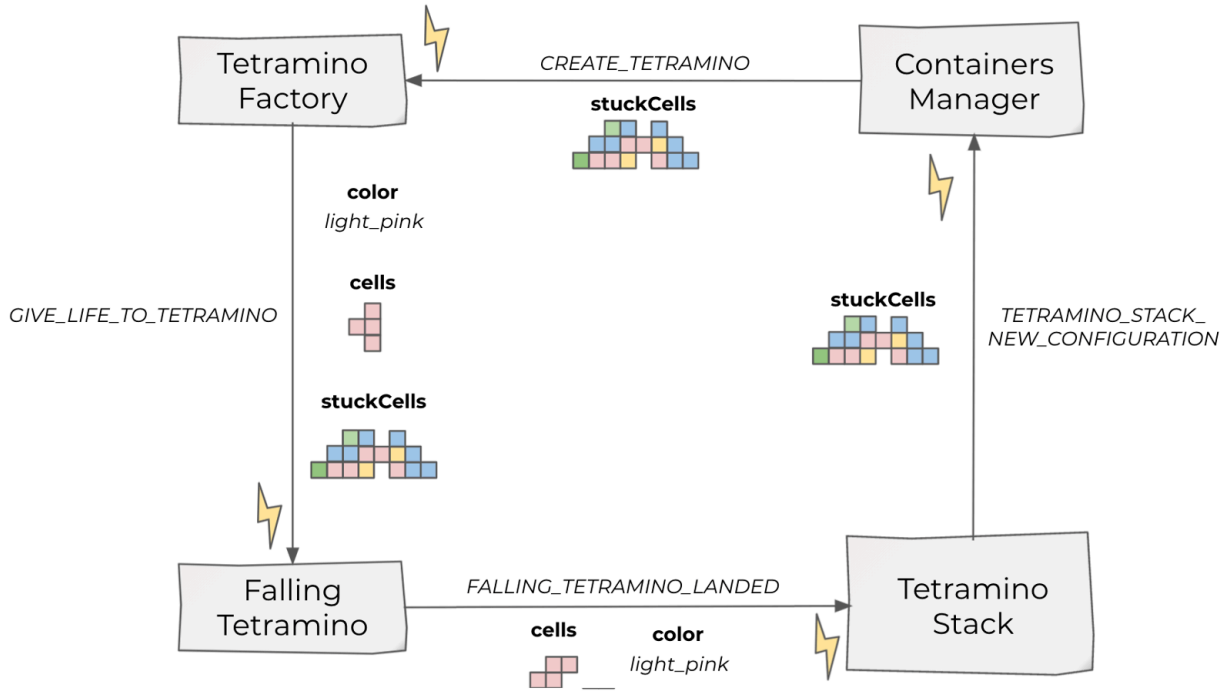

Figura 8.7: Mensagens cíclicas do jogo.

As condições que evitam um possível ciclo infinito e incontrolável (como um *livelock*, que veremos mais à frente no capítulo *Inconsistência distribuída*) ocorrem em:

- **FallingTetramino:** o serviço só emite o evento `FALLING_TETRAMINO_LANDED` quando o tetraminó ativo colide com uma das células estacionadas ou com o chão.
- **GameController:** só dispara o comando `CREATE_TETRAMINO` quando a altura da pilha de blocos não ultrapassa o limite do tabuleiro. Caso contrário, sinaliza o final do jogo emitindo `GAME_OVER`.

Esses são os pontos que considero cruciais para o entendimento desse estudo. Recomendo fortemente que clonem o repositório e executem o projeto na sua própria máquina se quiserem entender mais profundamente. Como dito anteriormente, existem classes e trechos que não foram mostrados por possuírem menor relevância, mas certamente outros pontos também poderão chamar a atenção.

8.3 E não é que funciona?

Surpreendentemente, pelo menos para mim, o experimento funcionou e o jogo é perfeitamente jogável. Contudo, existe um detalhe que deve ser examinado com um pouco mais de atenção. Ao analisarmos com mais atenção o fluxo de mensagens do componente `TetraminoFactory`, perceberemos que ele recebe dados para os quais não dá nenhuma finalidade a não ser repassá-los.

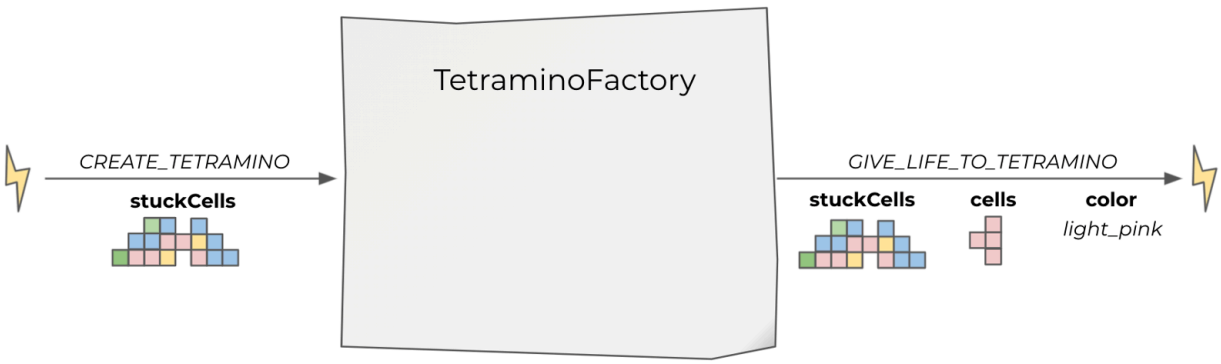

Figura 8.8: Dados repassados intactos em TetraminoFactory.

Note que o comando `CREATE_TETRAMINO` possui os dados `stuckCells` informando como está a configuração da pilha de células. Ao analisarmos a responsabilidade do comando (criar tetraminó) e a do componente que o recebe, **criar e exibir tetraminós**, veremos que essa informação não tem relação com nenhum dos dois. A única utilidade desse valor é ser enviado no comando encadeado `GIVE_LIFE_TO_TETRAMINO`, uma vez que o componente que o recebe, `FallingTetramino`, precisa dessa informação para calcular as colisões.

Existe um forte acoplamento implícito nessa comunicação. A presença desse atributo se dá porque `TetraminoFactory` sabe que a classe `FallingTetramino` escuta suas mensagens e que ela precisa dessa informação. Essa consciência quebra a vantagem inicial de que um componente não saberia da existência de outro. É até pior,

porque eles não só se conhecem como também sabem suas necessidades.

A presença de um atributo que não diz respeito ao domínio do próprio comando ou dos componentes envolvidos pode sinalizar um erro de design do projeto. Em aplicações pequenas, como é o caso, erros como tais são desprezíveis. Em aplicações maiores, por outro lado, podem ser apocalípticos. No melhor dos casos, geram no mínimo desperdício de rede e processamento ao retransmitir a mesma informação sem necessidade. Em outros casos, brechas de segurança podem ser introduzidas comprometendo todo o sistema e só deus sabe o que mais pode acontecer no pior dos cenários.

Mau cheiro de design

Por sinalizar um erro em potencial e não necessariamente um erro propriamente dito, essa característica me lembra, de alguma forma, o que Martin Fowler, no livro *Refatoração: Aperfeiçoando o design de códigos existentes*, chamou de **mau cheiro de código**: qualquer característica no código-fonte de um programa que possivelmente indica um problema mais profundo.

Ao expandir o conceito ao nível de abstração do design de microsserviços, podemos chamar o mesmo problema de **mau cheiro de design**: estruturas no design que indicam violação dos princípios fundamentais do design e impactam negativamente a sua qualidade.

Como exemplo de princípio de design violado na abordagem elaborada anteriormente cito o princípio do **aberto/fechado** adaptado à arquitetura de microsserviço descrito no capítulo *Microsserviços e a Orientação a Objetos*: *[...] um microsserviço nunca deve ter que ser modificado para fornecer funcionalidades que não respeitem o domínio apenas pela conveniência[...]*.

Uma solução seria fazer com que o componente `FallingTetramino` começasse a escutar também a mensagem

`TETRAMINO_STACK_NEW_CONFIGURATION`, enviada por `TetraminoStack` ao fim de cada rodada. Desse modo, o valor de `stuckCells` poderia ser eliminado do corpo das mensagens `CREATE_TETRAMINO` e `GIVE_LIFE_TO_TETRAMINO`.

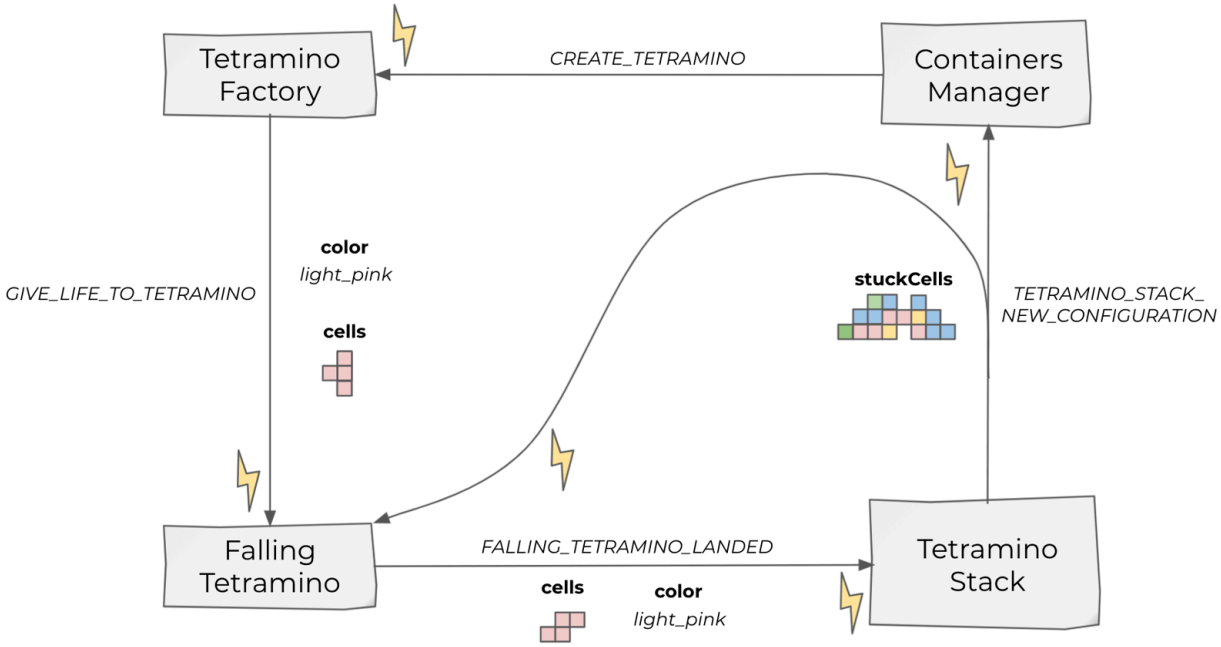

Figura 8.9: Resolve o mau cheiro, mas remove causalidade entre as operações.

Essa solução, como um palito de fósforo acendido no momento oportuno, removeria o mau cheiro. No entanto, removeria também um vínculo causal entre as mensagens (veremos mais sobre operações causais no capítulo *O inferno da consistência*). Observe, na abordagem anterior, que `FallingTetramino` inseria o tetraminó no tabuleiro ao receber a mensagem `GIVE_LIFE_TO_TETRAMINO`. Se continuarmos com a mesma lógica na nova abordagem e a mensagem `TETRAMINO_STACK_NEW_CONFIGURATION` se atrasar para chegar em `FallingTetramino`, o tetraminó cadente pode colidir com células inexistentes e/ou atravessar células que deveriam barrá-lo.

Atrasos de mensagens não são problemas reais na OOOMR por ser uma aplicação implementada com apenas uma thread, mas devemos lembrar que os atrasos (ou problemas similares, como reordenação

e falhas na entrega) são muito comuns em arquiteturas de microsserviços, que é o que a OOOMR tenta simular.

Felizmente, existem técnicas que podem ser aplicadas para sanar esse problema. Nesse cenário, especificamente, podemos fazer com que `FallingTetramino` espere as duas mensagens relacionadas (`GIVE_LIFE_TO_TETRAMINO` e `TETRAMINO_STACK_NEW_CONFIGURATION`) para efetivamente adicionar o tetraminó ao jogo. Para tal, podemos adicionar um identificador em ambas as mensagens e o componente `FallingTetramino`, ao receber uma das mensagens, deve obrigatoriamente esperar pela chegada da outra correspondente, ou seja, daquela que possui o mesmo identificador, para dar continuidade ao seu fluxo.

Se uma das mensagens demorar mais que um **tempo limite** para chegar, `FallingTetramino` pode solicitar o reenvio dela ou de ambas para que a fluidez da jogabilidade do jogo não seja impactada. Tempo limite, como veremos mais à frente, é um padrão de resiliência. Veremos esse e outros padrões com mais profundidade no capítulo *Fluxos resilientes*.

8.4 Resultado do experimento

Na abordagem OO tradicional, sem as três premissas elaboradas, seria mais intuitivo fazer com que as classes de negócio conhecessem umas às outras. A `FallingTetramino` poderia, por exemplo, requisitar (*get*) as posições de todas as células empilhadas de `TetraminoStack` para verificar a colisão cada vez que o tetraminó cadente descesse um nível, assim como a classe que verifica os comandos do usuário poderia conhecer a instância da classe `FallingTetramino` para executar uma ação diretamente e invocar algum método público com uma assinatura parecida com `moveRight()` ou `rotateClockwise()`.

Não é o que acontece na abordagem reativa. Sem o conhecimento uns dos outros, os componentes podem ser verdadeiramente autônomos e flexíveis como uma bailarina contorcionista que faz bico de ginasta nos finais de semana. Se um serviço estiver fora do ar, os outros funcionam perfeitamente, evitando catastróficas reações em cadeia.

Após o desenvolvimento, eis os insumos com uma opinião minimamente fundamentada.

Seja um bom ouvinte

Quanto mais cedo o objeto se subscrever nos tópicos, melhor. Assim, minimizam-se os riscos de um evento ser lançado sem que haja alguém para escutá-lo. Desse modo, o objeto tem o direito da escolha se vai reagir ou não ao receber a mensagem, direito que não lhe é garantido caso uma mensagem seja enviada antes que o objeto informe seu interesse nela. No nível de abstração dos microsserviços, para evitar esse cenário, algumas implementações de protocolos de comunicação possuem propriedades de persistência para que seja possível recuperar mensagens que foram enviadas e não foram escutadas. Casos do *Kafka*, *amqp*, *mqtt* e muitos outros.

Padrões de projeto

Existem técnicas para contornar as regras. É nesse momento que o jeitinho brasileiro traz vantagens. Por exemplo: a regra 1, a que abomina *getters*, pode ser contornada assim: *A* precisa dos dados *D* do objeto *B*? Então *A* se inscreve em um tópico sobre *D*, e toda vez que *D*, em *B*, for alterado, *B* publica uma mensagem e *A* é informado. Reatividade no seu mais puro néctar. 💔

Na parte sobre *Resiliência*, veremos outros padrões típicos de aplicações reativas, especialmente as voltadas à resiliência da

arquitetura.

Importância gramatical

De maneira geral, os nomes dos eventos são constituídos de, no mínimo, um verbo e um substantivo. Perceba:

1. *CREATE* (verbo) + *TETRAMINO* (substantivo).
2. *FALLING* (adjetivo) + *TETRAMINO* (substantivo) + *LANDED* (verbo).
3. *INPUT* (substantivo) + *DETECTED* (verbo).

Ao analisar esse padrão, notamos que a conjugação do verbo desempenha um papel importante na modelagem dos dados.

No primeiro exemplo, o verbo se apresenta no infinitivo/imperativo *CREATE* (criar, essa aparente indecisão se dá por conta da tradução do inglês para o português), os dois últimos se distinguem do primeiro, porque usam o passado/particípio (*LANDED* e *DETECTED*, aterrissou/aterrissado e detectou/detectado).

No livro, na parte *Orientação a Mensagens*, entenderemos que o primeiro se trata de um comando e os dois seguintes de eventos. Aguardem e confiem no pai.

Outro fator importante na conjugação é que, quando no passado, o verbo indica que algo já aconteceu, portanto não pode ser revertido – *De Volta Para o Futuro* e *O Exterminador do Futuro* discordariam disso –, ao passo que o infinitivo/imperativo, nesse caso, sinaliza uma ordem, que pode ou não ser seguida.

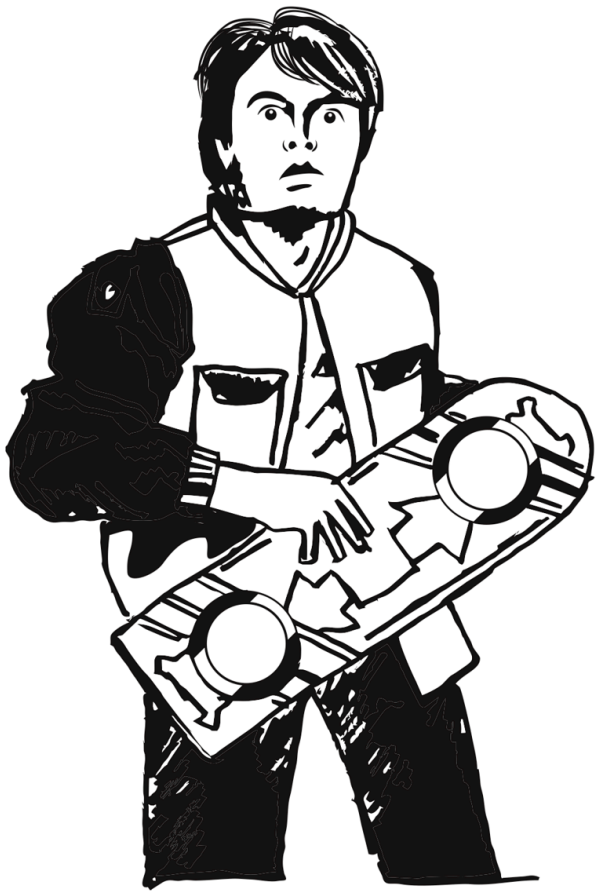

Figura 8.10: Great Scott.

Além disso, o tempo verbal, nesse contexto, atua como um indicador implícito de posse e, consequentemente, sinaliza o domínio do recurso. Quando a primeira mensagem, `CREATE_TETRAMINO`, no imperativo, ordena a criação de um novo tetraminó, o comando solicita que o ouvinte dê seus pulos para atender à ordem. Ou seja, o serviço que recebe o comando criará a peça e o recurso será dele.

Nos demais exemplos, `FALLING_TETRAMINO_LANDED` e `INPUT_DETECTED`, com os verbos no passado/particípio, os emissores indicam que a peça foi empilhada e que houve um comando do usuário. Portanto, quem possui o recurso, qual peça e comando, são os próprios emissores, cabendo aos receptores apenas a reação e a tomada de decisão baseadas nesse evento.

Desnormalização de dados

As mesmas informações são encontradas em diversos serviços, mas possuem significados e finalidades diferentes. As células empilhadas, por exemplo, são importantes para 1) o serviço que controla o jogo, `GameController`, saber se o fim da partida ocorreu; 2) o `FallingTetramino`, que controla o bloco ativo, detectar colisões; e 3) a classe que gerencia a pilha de células, `TetraminoStack`, verificar a eliminação das linhas ao pouso de uma peça.

O mesmo valor, replicado e reinterpretado por conveniência, tem o intuito de evitar transações e respeitar as regras para a OOOMR. Essa abordagem minimiza o acoplamento, o que é bom, mas, para tal, destaca-se a importância de manter os valores atualizados e consistentes entre si. Caso contrário, o jogo pode virar uma zorra, e os tetraminós utilizados para detecção de colisão, controlados por `FallingTetramino`, poderão ser diferentes dos tetraminós suscetíveis à eliminação gerenciados por `TetraminoStack`.

Já viu a bagunça que isso pode dar, né? Peças invisíveis bloqueando o bloco cadente e jogo sendo finalizado de maneira incompreensível, para dizer o mínimo. Falo com tranquilidade, até porque caí nessa armadilha mais vezes do que me orgulho durante o desenvolvimento.

Fonte única de emissão

Como há similaridade entre as finalidades de algumas mensagens, é tentador fundi-las e fazer com que seus remetentes diferentes enviem a fusão. Seguindo o princípio *Don't Repeat Yourself* (não se repita), é natural pensar em dois remetentes enviando a mesma mensagem, mas com conteúdos diferentes. É como dizem: todo problema tem uma solução simples, rápida e extremamente errada.

Essa não é uma boa ideia, porque, ainda que os conteúdos das mensagens fossem idênticos, elas possuiriam finalidades distintas. A fusão é uma bomba-relógio. Eventualmente, as finalidades exigirão

atributos e/ou momentos de disparo diferentes. Fazer uma alteração tardia desse porte demandará um rastreamento potencialmente exponencial para saber quem consome, quando e para quê. É mais sensato fazer com que um evento seja disparado apenas por um componente.

Para exemplificar, imagine que queiramos contabilizar pontos nas partidas. Para cada peça empilhada, 10 pontos e, para cada linha eliminada, 100 pontos. Pode ser intuitivo fazer com que os controladores desses recursos emitam um novo comando `ADD_SCORE` com a pontuação adquirida pelo evento. Assim, `FallingTetramino`, quando empilhasse a peça, emitiria o comando indicando a adição de 10 pontos, e `TetraminoStack`, ao perceber que uma linha foi eliminada, lançaria a mensagem informando a necessidade de adicionar 100 pontos à pontuação corrente. O mesmo comando, `ADD_SCORE`, emitido por duas entidades diferentes. Ambas as mensagens sendo capturadas por uma nova classe hipotética, `ScoreHandler`.

Observem que o conceito de pontuação, até então inédito nesse jogo, não é de responsabilidade de `FallingTetramino` e tampouco de `TetraminoStack`.

Sob essa ótica, o mais apropriado seria que a entidade criada com esse propósito, `ScoreHandler`, capturasse os eventos já existentes, `FALLING_TETRAMINO_LANDED` e `LINE_DELETED`, e incrementasse o atributo que representa a pontuação corrente. Só então, se julgar necessário, ela mesma emitiria o evento `SCORE_ADDED` com a pontuação adicionada. Nesse cenário, a mensagem `SCORE_ADDED` seria emitida apenas por um serviço e os princípios mencionados seriam respeitados. Todo mundo sai ganhando e não há pânico.

Definir o emissor do evento é uma tarefa árdua. Em muitos momentos, há vários candidatos e a escolha não fará diferença. No entanto, existem situações em que uma opção malfeita exigirá redesenho e retrabalho de urgência. Ninguém gosta disso,

principalmente porque, segundo meus cálculos, esse tipo de coisa costuma acontecer no fim do expediente da sexta-feira.

Para exemplificar a dificuldade de definir o emissor, pergunto: quem deveria indicar a criação de um novo bloco? O tabuleiro? O controlador do bloco ativo? Uma nova entidade? Honestamente, eu realmente não sei. Provavelmente, a resposta imediata é "tanto faz" e conseguiríamos ótimos argumentos para justificá-la.

Apresentaremos formalmente o princípio do **Escritor Único** e o impacto dessa escolha na consistência dos dados na parte *Elasticidade*.

Mapeamento dos fluxos é fundamental

Desenhar fluxos pode parecer fácil em um primeiro momento. Contudo, redesenhá-los é uma tarefa constante e difícil. Rastrear os fluxos dos eventos é ainda mais difícil. De longe, é o maior desafio que encontrei. Quem emite o quê, quando e por quê? Em larga escala e em uma arquitetura distribuída, como uma de microsserviços reativa, ferramentas para rastrear as mensagens são imprescindíveis.

Por mais esbeltos e coloridos que sejam, não foi fácil gerar os diagramas do capítulo. Principalmente após algumas dezenas de alterações. Chegou a um ponto em que os diagramas ficavam tão confusos e se mexiam tanto que pareciam as escadarias de Hogwarts.

Na parte *Resiliência*, abordaremos a **observabilidade**, assunto que será discutido com maior profundidade.

Aparência de smalltalk

Guardadas as diferenças, especialmente sintáticas, os códigos produzidos guardam lógicas semelhantes com as produzidas pela linguagem de programação **smalltalk**, linguagem que, por definição, tem a passagem de mensagens como a única forma de

comunicação entre os objetos. Nela, é impossível que dois objetos acessem as variáveis de instância um do outro, mesmo que pertençam à mesma classe.

Alan Kay, citado anteriormente como um dos criadores da OO e que também colaborou na criação de smalltalk, reconhece (e explica em https://www.quora.com/What-did-Alan-Kay-mean-by-I-made-up-the-term-object-oriented-and-I-can-tell-you-I-did-not-have-C++-in-mind) que, quando cunhou o termo OO, ele não tinha C++ em mente. Ou seja, para o criador da OO, linguagens, como C++ e similares distorceram a real ideia da OO e smalltalk fazia bom uso dela.

> "A contribuição de smalltalk é um novo paradigma de design – que eu chamo de Orientação a Objeto." - Alan Kay.

Um artigo muito interessante sobre essas afirmações e suas implicações foi escrito por Curtis Poe e pode ser encontrado em https://ovid.github.io/articles/alan-kay-and-oo-programming.html. Leitura mais do que recomendada.

Se essa semelhança é algo positivo ou negativo, eu realmente não sei, mas supus que fosse válido comentar.

Diferentemente de smalltalk, na OOOMR, as mensagens não têm destinatários definidos. Voltaremos a falar sobre o que isso significa na parte *Orientação a Mensagens*.

Acoplamento fraco e atemporal

As instâncias são fracamente acopladas, porque não têm conhecimento umas das outras. Não sabem sequer que existem. Elas também não possuem vínculo temporal explícito, porque cada uma faz o que lhe cabe no momento em que lhe é conveniente e confia que as outras também o farão.

Esse desacoplamento temporal abre uma verdadeira abundância de possibilidades para a arquitetura reativa. Como veremos mais à frente, isso viabiliza o controle de vazão das mensagens de forma que não impacte na conclusão da operação. O desacoplamento permite o agrupamento de mensagens (no futuro, daremos o nome de **processamento em lotes** ou **associatividade** a essa técnica) e também permite a técnica de **contrapressão**.

Ao desacoplar temporalmente, minimizamos os riscos de uma catástrofe em cadeia quando um serviço se engasga com um processamento e acaba afetando os serviços adjacentes com sobrecargas.

Em jogos, essa característica pode não ser tão útil. Intrinsecamente, espera-se uma relação temporal entre o comando do jogador e o movimento da peça. O intervalo de tempo é determinante para o sucesso e a jogabilidade da jogatina.

Quantos e quantos controles, consoles, laptops e monitores já não foram quebrados por conta de um suposto *lag* no jogo? Observação: não me referi a ninguém especificamente. Qualquer semelhança com alguém com quem eu possa ter trabalhado na vida não passa de uma mera coincidência.

Ainda assim, ao extrapolarmos essa qualidade para outros contextos, conseguimos perceber a utilidade da técnica.

8.5 Conclusão

A quantidade de insumos produzidos por esse projeto de prova de conceito foi colossal. Sugiro que reproduzam o estudo com outros projetos similarmente simples, como o jogo da nave e o da cobrinha, e compartilhem as experiências. Certamente haverá outras, o que torna a atividade válida e repleta de aprendizado.

A diversão é certa, e vocês apreciarão eternamente. Alerto de antemão que jogá-los após implementá-los pode ser mais viciante do que assistir àqueles tutoriais do Rodrigo Hilbert indiano no YouTube, especialmente o jogo da cobrinha, tinha esquecido o quão apaixonante ele era – o jogo, não o Rodrigo Hilbert indiano.

Figura 8.11: Games antigos, saudades.

Parte 4: Orientação a Mensagens

Capítulo 9
Mensagens distribuídas

Em um sistema distribuído, a mensagem desempenha um papel insubstituível. Os dados sempre fluem de um componente para outro por meio de troca de mensagens. Sem ela, a distribuição seria, literalmente, impossível. Protocolos e formatos diferentes são usados para compartilhar dados em um sistema distribuído como mensagens.

O Manifesto Reativo afirma que os sistemas reativos devem se basear em trocas de mensagens assíncronas para estabelecer uma fronteira entre os componentes. É justo que exploremos mais sobre o conceito de mensagens.

Figura 9.1: Onde está o Wally?

9.1 Conceito

De maneira geral, é assim que a comunicação ocorre em um componente: 1) recebe uma mensagem, 2) interpreta e 3) responde. Este último é opcional e, cá entre nós, deveria ser menos utilizado em muitas situações.

A **teoria da comunicação** estuda o processo científico de enviar e receber informações. Existem muitos princípios, métodos e componentes que podem afetar uma mensagem e a teoria da comunicação explica isso tudo.

Como a comunicação se tornou tão complexa com o tempo, existem diferentes modelos de comunicação para diferentes categorias de comunicação. São muitos modelos, mas no geral, eles costumam formalizar e identificar os seguintes agentes: **emissor**, **receptor**, **código**, **canal**, **ruído**, **resposta** e, por último e foco deste capítulo, a **mensagem**, o objeto da comunicação, constituído pelo conteúdo das informações transmitidas.

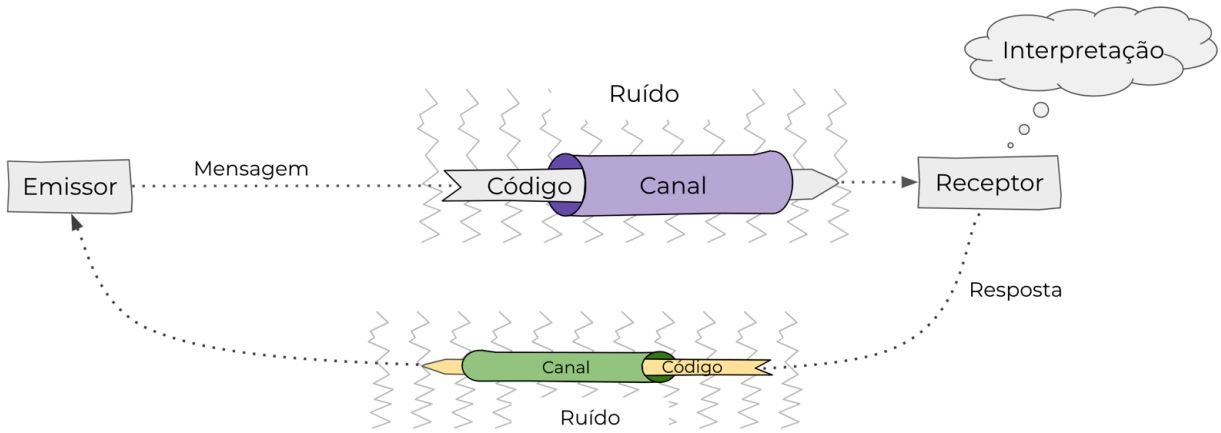

Figura 9.2: Elementos comuns nos modelos de comunicação.

Em inglês: *message*, em grego: *mínyma*, em latim: *strategi*, em francês: *stratégie*. A mensagem é um conceito, uma informação, um dado presente na comunicação. Ela é a unidade básica de comunicação e pode ser literalmente qualquer coisa. Um

identificador, um texto, um *sticker* no WhatsApp ou o inevitável sinal de negativo durante um encontro amoroso.

A mensagem, por si só, não tem nenhum propósito e não é reveladora. Não ter nenhuma intenção especial torna as mensagens genéricas, mas também menos significativas. Contudo, quando a desmembramos, a inserimos em um contexto e analisamos suas partes constituintes, entendemos suas intenções.

9.2 Decomposição de mensagens

Nos sistemas reativos, as mensagens são subdivididas em três grupos. De modo superficial, os grupos são **comandos**, mensagens que dão ordens; **eventos**, noticiam fatos e as **consultas** solicitam o estado atual de algo.

A missão deste capítulo é desembaraçar o sarapatel de termos em negrito do parágrafo anterior e aprofundar a distinção e a função de cada um deles.

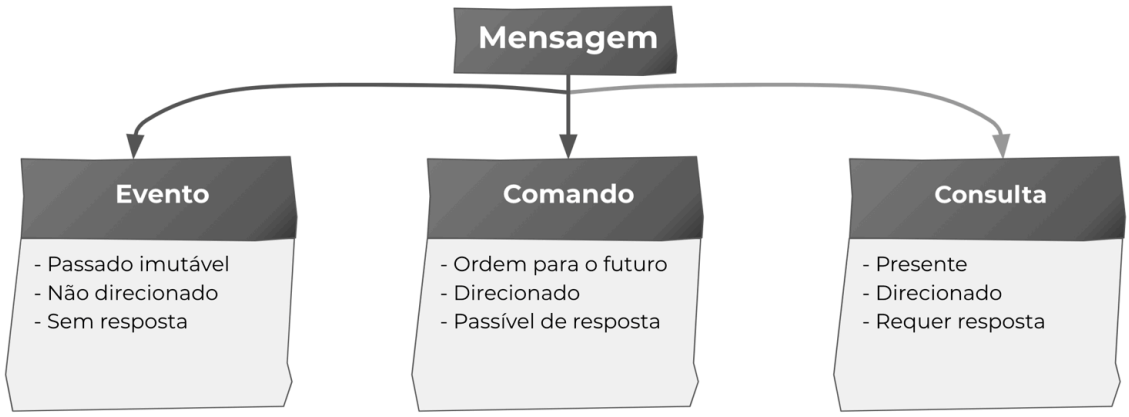

Figura 9.3: Com quantos paus se faz uma mensagem?

Eventos

Um evento é uma mensagem que informa vários ouvintes sobre algo que aconteceu, uma **mudança relevante de estado**. Ele é enviado por um produtor, que não conhece e não se importa com os consumidores do evento. Como os eventos aconteceram no passado, eles são nomeados usando o particípio/passado.

Na sua forma mais básica, um componente ou serviço provoca um evento, e outros reagem a ele dependendo de seu interesse no referido evento. Ou, como diz a juventude do Twitter hoje em dia, um componente dispara o evento e os outros que lutem para lidar com isso. Nada acontece após a emissão do evento, a menos que serviços ou componentes interessados nos eventos em questão sejam configurados para reagir ou responder.

A emissão do evento `MENSAGEM_ENVIADA` pode ser representada pelo surgimento de um pequeno ✓ cinza.

O disparo do evento `MENSAGEM_ENTREGUE` pode significar o aparecimento de outro pequeno ✓ cinza, e o evento `MENSAGEM_LIDA`, além de fazer com que os dois ✓ fiquem da cor azul, pode desencadear uma possível crise de ansiedade.

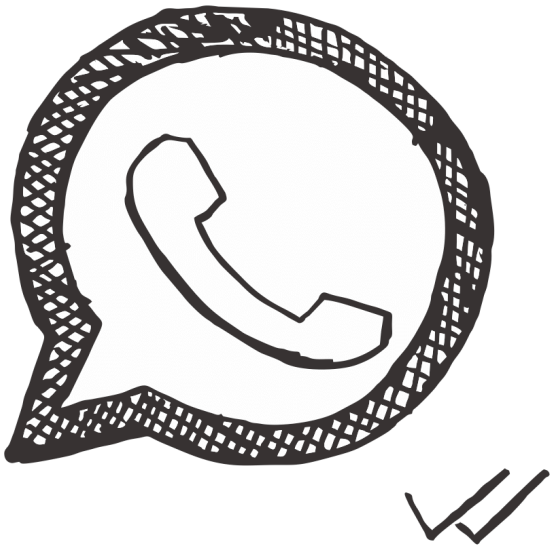

Figura 9.4: Reação visual aos eventos recebidos.

Comandos

Um comando representa algo que é pretendido ou **solicitado que aconteça**. É uma conexão direta entre um remetente, que envia o comando, e um destinatário, que recebe o comando. O verbo costuma estar no imperativo/infinitivo, porque há um intuito de ordem.

Pelo fato de o termo "comando" ser amplamente usado em outros contextos, Martin Fowler sugere a utilização de "modificadores" ou "mudadores" (https://martinfowler.com/bliki/CommandQuerySeparation.html). A nomenclatura apropriada ficará a seu critério, caro leitor ou leitora.

Os comandos podem significar uma solicitação de mutação de estado. Por exemplo, ao perceber a emissão do evento `MENSAGEM_ENVIADA`, o componente pode enviar o comando

`REORDENAR_LISTA_DE_CONVERSAS`, que alterará a ordem de disposição das conversas no aplicativo, fazendo com que as conversas com mensagens mais recentes fiquem no topo.

Figura 9.5: Evento como resultado de um comando.

Os comandos podem levar a eventos, especialmente se alguma mudança de estado tiver ocorrido. O evento `LISTA_DE_CONVERSAS_REORDENADAS` pode ser o resultado do tratamento de um comando `REORDENAR_LISTA_DE_CONVERSAS`.

Após o evento `LISTA_DE_CONVERSAS_REORDENADAS`, poderíamos ter disparado o evento `ATUALIZAÇÃO_DE_TELA_NECESSÁRIA`. É muito comum encontrar eventos com esse perfil. Mas, na verdade, isso não é realmente um evento. É um comando disfarçado de evento. Um evento normalmente permite que alguém saiba que algo aconteceu. É uma transmissão e você não se importa com quem a captura.

Quando você emite um comando, como `ATUALIZAÇÃO_DE_TELA_NECESSÁRIA`, você não está dizendo ao mundo que algo aconteceu, mas que deseja que o atualizador da tela faça algo para você. Seria melhor tornar isso explícito com um comando de `RENDERIZAR_TELA`.

Consultas

Solicitar o estado atual de um serviço ou um aspecto dele é chamado de consulta. São operações que recuperam informações

dos dados na aplicação para atender a uma necessidade e, por isso, elas costumam contrariar o princípio *Tell, don't ask*. As consultas não causam alteração de estado ou qualquer outro efeito colateral e exigem respostas subsequentes, mesmo que vazias.

Um exemplo de consulta é aquela pergunta que o Burro Falante do Shrek fazia exaustivamente na história que contamos há alguns capítulos: "A gente já chegou?".

As consultas geralmente carregam um nome que reflete a expectativa de obtenção de um resultado. Por exemplo, alguns verbos que explicitam essa intenção e são tipicamente usados nas consultas são `RECUPERAR_INFORMAÇÕES` (**recuperar**), `LISTAR_DADOS_DO_USUÁRIO` (**listar**), `BUSCAR_DISPOSITIVOS_ASSOCIADOS` (**buscar**) ou `OBTER_RESPOSTA_DO_CONVITE_DO_CINEMA` (**obter**).

Uma implementação bastante comum de uma consulta, e meu desafeto pessoal, é o método *GET* do http. Apesar de o método, teoricamente, representar uma consulta, ele nem sempre o é - o que faz com que eu nutra um ódio ainda maior por ele. Relevo, porque o ódio é um sentimento genuíno e às vezes é o único sentimento possível.

Apesar de, intuitivamente, associada a um comportamento síncrono, a consulta não tem essa implicação. Ainda assim, nos sistemas reativos, por sua Orientação a Mensagens assíncronas, as consultas não são bem-vistas.

Por conta disso, os sistemas reativos transformam a consulta na união de um comando com um evento. Há o envio de um comando indicando uma necessidade e, posteriormente, um evento é lançado com uma resposta. Por exemplo, o comando `CONVIDAR_PARA_O_CINEMA` seguido do infalível evento `CONVITE_RECUSADO_COM_LOUVOR` .

Para associar mensagens, como o evento ao comando da substituição da consulta, Gregor Hohpe e Bobby Woolf, em *Enterprise Integration Patterns*, falam sobre identificadores de sessão e de correlação (você pode ler mais a respeito aqui

https://www.enterpriseintegrationpatterns.com/patterns/messaging/CorrelationIdentifier.html).

Esses indicadores são usados para fazer um mapeamento e rastrear qual mensagem se refere a qual. Se o identificador for o mesmo, há uma relação entre as mensagens e a espera da resposta pode ser encerrada.

Algumas literaturas consideram a resposta da consulta como outra subcategoria de mensagem. É compreensível, porque também se trata de uma unidade básica de informação. Ainda assim, ela é bastante similar a um evento, sendo distinta de tal apenas por existir uma consulta anterior.

Como essa consideração não tem valor prático e como três subdivisões são mais fáceis de memorizar do que quatro, prefiro concordar com as literaturas que afirmam só haver três categorias de mensagem.

9.3 CQRS

Essas divisões conceituais podem parecer bobeiras insignificantes. Tipo aquela galera que diz que há diferença entre Heineken e Brahma apenas para soar mais inteligente ou pedante – ou os dois. No entanto, é exatamente nessa distinção que o **CQRS** (*Command Query Responsibility Segregation*, traduzido levianamente como Segregação de Responsabilidades de Consulta e Comando) mostra sua cara.

O CQRS propõe a utilização de um modelo para atualizar informações e outro para lê-las. Como as operações de leitura e gravação costumam ser assimétricas, com requisitos de desempenho, consistência, disponibilidade e escalabilidade muito diferentes, o CQRS permite que as operações sejam tratadas de

formas diferentes, otimizadas e mapeadas em seus modelos ideais, provavelmente rodando até em hardwares separados.

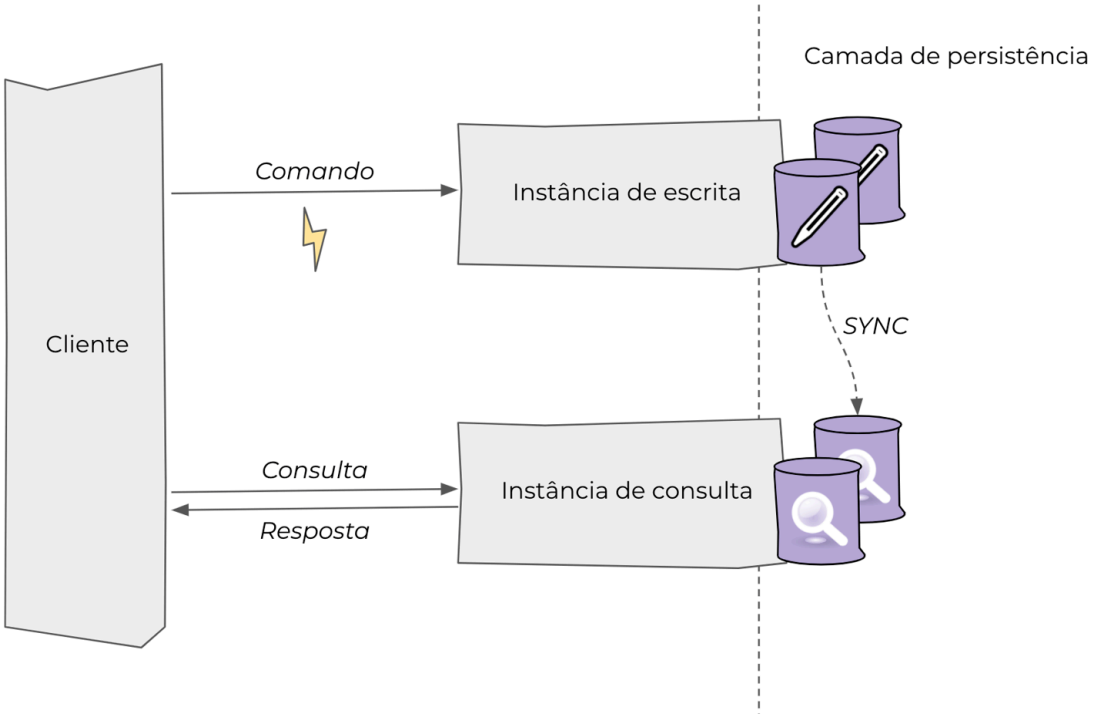

Figura 9.6: Operações segregadas.

Convém ressaltar, no entanto, que é incomum que um problema esteja apto a ser resolvido com essa técnica. Normalmente, há interseção suficiente entre os lados do comando e da consulta para que seja mais fácil compartilhar um modelo, e CQRS adicionaria complexidade e reduziria a produtividade.

9.4 Orientação a Mensagens vs Orientação a Eventos

Em arquiteturas Orientadas a Mensagens, os serviços produzem mensagens indicando que devem ser entregues a um endereço específico.

Em arquiteturas Orientadas a Eventos, uma vez que eventos não são direcionados, os componentes declaram apenas onde anunciarão seus eventos. Portanto, um componente que emite eventos usará um local conhecido para publicá-los, mas não saberá quais componentes estão consumindo os eventos.

Essa comparação, entretanto, pode ser enganosa. O termo "Orientado a Mensagens" se refere a um bloco de construção em um sistema enquanto "Orientado a Eventos" se refere a uma propriedade de nível superior de um sistema. Portanto, como eventos são uma subcategoria de mensagens, ao usar ferramentas Orientadas a Mensagens, podemos construir sistemas Orientados a Eventos.

9.5 Conclusão

O envio de mensagens nos sistemas reativos enfraquece o acoplamento. O receptor e o emissor não precisam estar disponíveis ao mesmo tempo para que haja comunicação. Eles nem sequer precisam saber da existência um do outro. Grosso modo, as mensagens são divididas em três categorias: eventos, consultas e comandos.

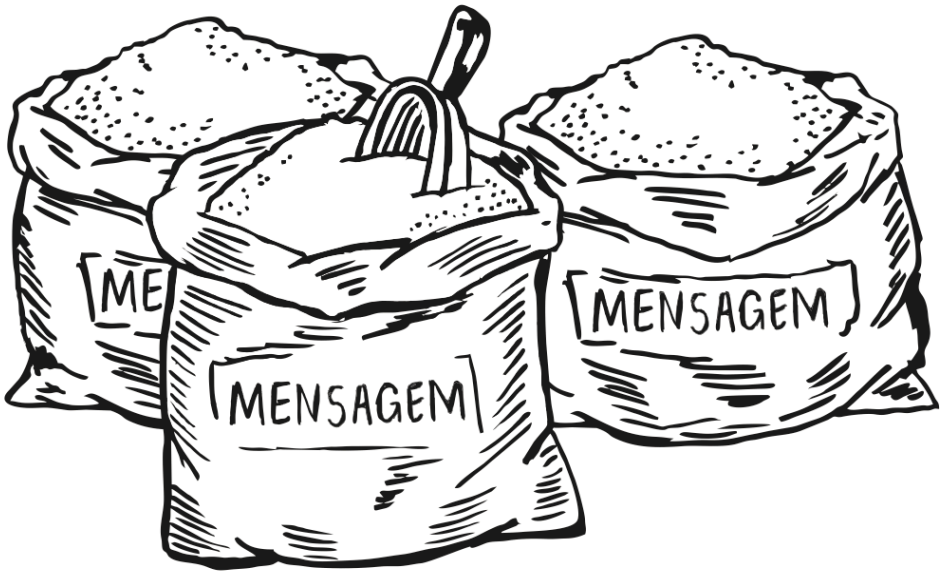

Figura 9.7: Farinhas do saco de mensagem.

Eventos, consultas e comandos são farinhas do mesmo saco, cuja embalagem possui, em letras garrafais, um rótulo escrito: MENSAGEM.

Esses são mais alguns dos ingredientes da sopa de letrinhas que constituem os sistemas reativos e devem ser adicionados ao vocabulário do desenvolvedor e da desenvolvedora para serem fermentados e criar um design mais apropriado.

Como é sabido pelos amantes do pó rico em amido, as farinhas são extremamente versáteis e são usadas em receitas que conseguem agradar a todos os paladares.

Além dos sistemas distribuídos e da minha adorada tapioca, existe uma infinidade de produtos gastronômicos elaborados a partir da farinha, como bolo de fubá, farofada de frango e pizza. Minha receita favorita é a de um excelente pavê de carne. Não tem mistério. Basta adicionar farinha, biscoito, geleia, creme, framboesa,

bife acebolado e ervilha, bater tudo no liquidificador e correr para o abraço.

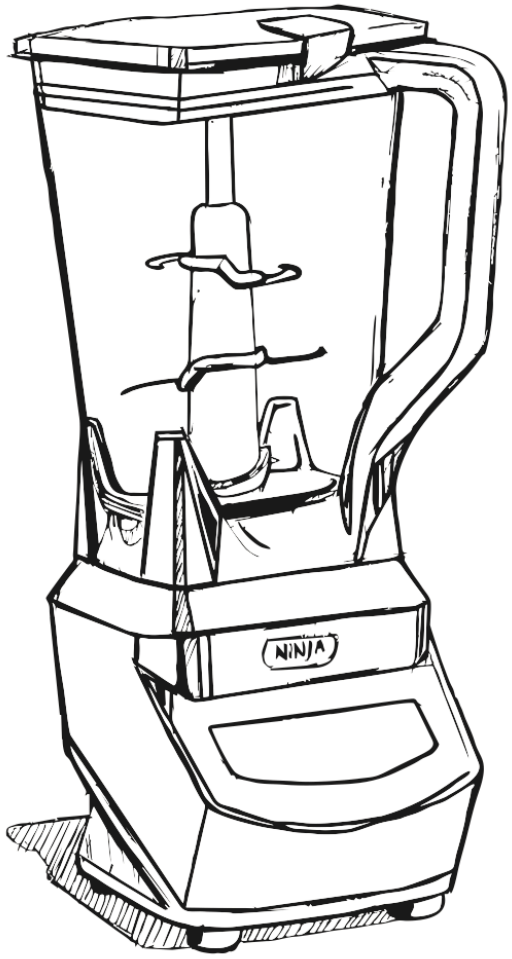

Figura 9.8: Sistemas reativos, não confundir com sistemas rotativos.

Como visto no Manifesto Reativo, é fundamental que o mecanismo de comunicação entre os componentes seja assíncrono. Isso desencoraja o uso de consultas síncronas nos sistemas reativos. Como dito, em vez dela, você pode usar a associação assíncrona de comandos e eventos para atingir o mesmo objetivo.

A natureza não bloqueante da comunicação assíncrona permite que os destinatários consumam recursos apenas quando estiverem ativos, o que acaba levando a menos sobrecarga do sistema.

Além disso, ela permite que os sistemas reativos estabeleçam uma camada entre os componentes que garantirá acoplamento fraco, isolamento, transparência de localização e fornece os meios para delegar erros como mensagens.

Ao desacoplar os componentes de um sistema reativo, podemos começar a implementar as outras características reativas definidas no Manifesto Reativo. Podemos ser resilientes ao ocorrerem falhas em certos componentes de nossa aplicação e podemos ser elásticos aumentando ou diminuindo os recursos dos componentes.

Capítulo 10
Controle de vazão

Mensagens são fundamentais por propagarem alterações entre vários microsserviços e seus modelos de domínio relacionados. O mecanismo de controle de vazão de mensagens controla a velocidade de transmissão das mensagens no fluxo para evitar que os microsserviços se sintam sobrecarregados ou ociosos.

Como os microsserviços devem ser independentes e autônomos, potenciais gargalos são recorrentes e evitá-los antes que explodam aprimorará a resiliência da arquitetura de microsserviços. Em outras palavras, o controle de vazão equilibra o fluxo e determina parte do desempenho.

Os fluxos de mensagens precisam ser apenas construídos com cuidado e a taxa na qual os dados estão se movendo precisa ser monitorada continuamente.

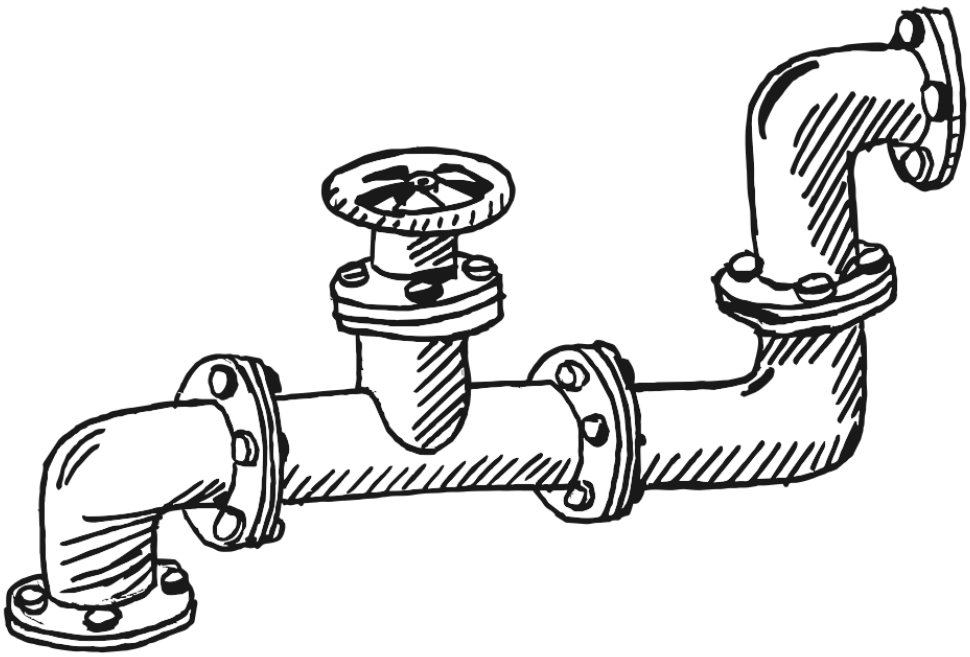

Figura 10.1: Controle de vazão.

Como exemplos de mecanismos de controle de vazão e vazão de mensagens, cito o **processamento em lotes** e a **contrapressão**.

10.1 Processamento em lote

Uma forma eficiente de processar mensagens é agrupá-las e processá-las em lotes. Nesse modelo, as mensagens são recebidas e agrupadas em um sistema até que processadas periodicamente.

O processamento em lotes oferece a oportunidade de dividir o custo de operações caras, como *I/O* ou cálculos caros. Por exemplo, empacotar vários itens de dados na mesma mensagem transmitida pela rede ou bloco de disco para aumentar a eficiência e reduzir a utilização. Outros argumentos são a otimização de consultas no banco de dados ou até mesmo a capacidade de remover automaticamente os dados mais antigos para que o desperdício de espaço com processamento redundante seja evitado.

A implementação dos sistemas de processamento em lotes possibilita que, quando o volume de dados aumentar, o tempo de processamento não aumente na mesma proporção. Uma das principais vantagens desse modelo é que o processamento pode ser feito na conveniência do receptor sem se preocupar com o instante de recepção do evento.

Para que o processamento em lotes seja possível, devemos garantir que o agrupamento de mensagens não impactará no resultado da tarefa. Isso pode ser obtido com o desacoplamento temporal existente em fluxos assíncronos. À habilidade de um fluxo ser capaz de agrupar mensagens sem impactar na conclusão da operação, como veremos mais à frente, damos o nome de **associatividade**.

O tamanho ideal do lote pode variar ao longo do ciclo de vida de um serviço. Valores, como CPU e memória utilizados do receptor, ou fatores externos, como o horário do dia, podem influenciar no tamanho apropriado do lote.

Uma das maneiras de comunicar o tamanho ideal do lote para o receptor naquele momento é enviar uma mensagem endereçada ao emissor, na forma de *feedback*, com essa informação.

10.2 Contrapressão

No modelo de comunicação mencionado no capítulo anterior, a contrapressão, também conhecida como *backpressure*, é a **resposta**, o *feedback*. É a reversão opcional do processo de comunicação em que o receptor se torna o emissor e o emissor se torna o receptor. Apesar do canal e o código da resposta não precisarem ser os mesmos da mensagem original, eles costumam ser.

Dado que é inadmissível que um componente sob estresse falhe catastroficamente ou se engasgue com mensagens de maneira

descontrolada, ele deve anunciar que está com dificuldades para os componentes que se comunicam com ele e, assim, fazê-los reduzir a carga. Esse mecanismo assegurará que o sistema tenha mais resiliência e elasticidade, porque fornecerá informações que podem distribuir melhor a carga ou ajustar o tamanho dos lotes.

A contrapressão é uma metamensagem e pode retroceder sucessivamente até o usuário, caso necessário. Momento em que a responsividade sofrerá degradação.

O recado da contrapressão é, normalmente, alertar que o sistema está sob estresse e que precisa de um tempo para respirar. Mas também pode ser o oposto, o sistema está suave na nave e quer uma injeção de adrenalina para acelerar o processamento de mensagens.

Figura 10.2: Acelera! Acelera, Jesus. Acelera, senhor.

Por ser uma mensagem contrária à original, meta atributos de protocolos síncronos, como os cabeçalhos http, facilitam sua implementação. Podemos, convenientemente, inserir informações de contrapressão na resposta da requisição. Essa implementação, contudo, limita a contrapressão apenas à instância que fez a

requisição, as demais necessitarão ser avisadas em outro momento oportuno.

Falando em http, há um código de status com a finalidade de identificar que a taxa limite foi ultrapassada: `429 - Too Many Requests` (algo que pode ser traduzido como "Muitas requisições"). Segundo uma de suas documentações, um cabeçalho `Retry-After` pode ser incluído na resposta indicando quanto tempo o usuário deve esperar antes de fazer um novo pedido.

Se enviada assincronamente, temporalmente desatrelada a qualquer requisição, alguma lógica deve ser imputada na mensagem para que todas as instâncias do emissor a receba e desconsidere reordenamentos de mensagens. Veremos na parte *Elasticidade* como garantir isso.

A informação adicional anexada à contrapressão não deve ser complicada. Ela deve ser desenvolvida de modo que possa facilmente ser acrescentada em quaisquer interações entre os microsserviços e não algo que precisa ser reinventado ou manualmente controlado a cada interação.

Uma forma conhecida, talvez a mais famosa, de contrapressão é o *throttling*. Essa implementação é um padrão de consistência de microsserviços e consiste no ato de se recusar a processar as mensagens por estar sobrecarregado.

Ele delega ao remetente da comunicação a tarefa de maneirar na frequência de mensagens ou utilizar mecanismos alternativos, como *fallback* e *circuit breakers*. Na parte *Responsividade*, explicaremos como esses padrões e outros funcionam.

Uma alternativa à contrapressão é dar mais poder computacional ao destinatário ou executá-lo em mais máquinas, ações conhecidas como **escalabilidade**. É necessário ter atenção para evitar que quando uma contrapressão informar que está ociosa e deseja mais carga de trabalho, o aumento de vazão resulte no acúmulo de mensagens no serviço seguinte no fluxo de mensagens.

10.3 Conclusão

Processamento em lote e contrapressão representam mecanismos de controle de vazão e atuam como válvulas represadoras e diluidoras de fluxo de mensagens.

Figura 10.3: Sistemas reativos, não confundir com sistemas representativos.

Eles possuem os mesmos propósitos: cortar desperdícios, regular a vazão e evitar falhas. Excessos, para mais e para menos, geram desperdício de recurso e podem resultar em falhas.

O primeiro evita a falha por propor economia de custos monetários e computacionais ao acumular mensagens para processamento quando poupa, entre outros, acesso ao disco e utilização de rede. O segundo indica como o serviço receptor está lidando com a vazão

atual. Ao sinalizar sobrecarga, o remetente pode diminuir o tamanho do lote de mensagens. Ao informar que está ocioso, o remetente pode aumentar a vazão e saciar o apetite *workaholic* do destinatário.

Jonas Bonér, criador do Akka e um dos autores do Manifesto Reativo, no seu livro *Reactive Microsystems: The Evolution of Microservices at Scale*, declara que a contrapressão é um recurso indispensável e deve ser sempre utilizado. Ainda segundo Bonér, a contrapressão aumenta a confiabilidade não apenas dos componentes individuais, mas também no fluxo de dados e de toda a arquitetura.

Capítulo 11
Mensagens poliglotas

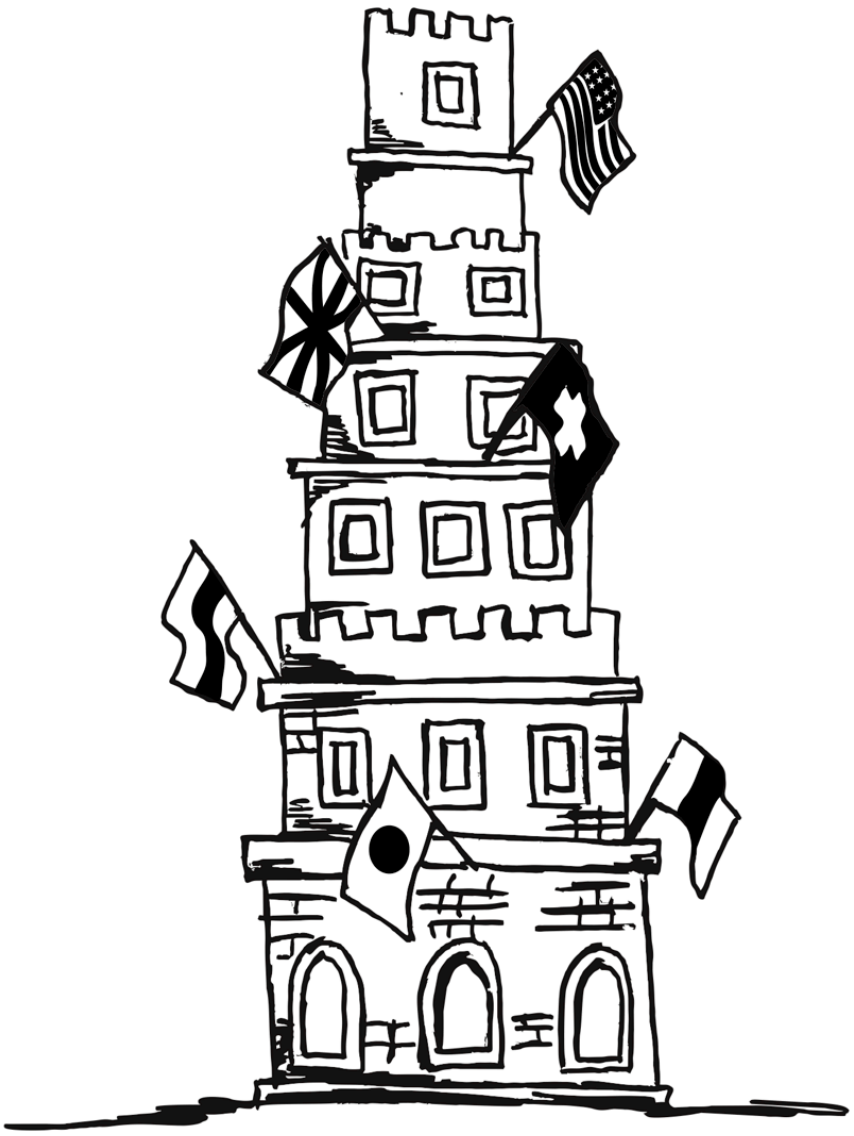

Figura 11.1: A torre de Babel da computação.

Ainda fazendo uso dos modelos da teoria da comunicação dos capítulos anteriores, o código desempenha papel indispensável para

a comunicação: "a maneira pela qual a mensagem se organiza". É uma coleção de regras que governa como a troca de informações se sucederá, um comum e predeterminado acordo entre o **emissor** e o **receptor**.

O código deste livro, por exemplo, consiste em um conjunto de sinais gráficos, as letras do nosso alfabeto, sob as leis do português. Este mesmo livro traduzido para outro idioma conteria a mesma mensagem, mas seria regulado por outro código.

Figura 11.2: Mesma mensagem, vários códigos.

Indo bem longe, analisando a forma como a comunicação na humanidade ocorre desde sempre, identificamos uma miríade de códigos verbais, visuais, defumados, luminosos, acrobáticos, discos da Xuxa tocados ao contrário, mediúnicos e muitos outros. Existem várias maneiras de passar uma mensagem.

Os motivos para abundância são incontáveis, mas geralmente envolvem restrições do meio, do emissor e/ou do receptor. Dependendo dessa combinação, alguns códigos são mais adequados

e outros são impossíveis. Já tentou ter uma conversa verbal debaixo d'água? Experimente conversar gestualmente em ambientes escuros ou entregar instruções por escrito ao seu adorável pet. Por mais que eu tente, Ricardo Severus, meu gatinho, se limita a interpretar tudo como "seja cinza".

Libras, por exemplo, é um código voltado para a comunicação de pessoas com deficiência auditiva. Entretanto, exige luminosidade e contato visual. O código Morse, por sua vez, demanda apenas um sinal, sendo este longo ou curto para o envio de mensagens.

Stickers e memes são excelentes exemplos, mas só pessoas profundamente letradas são capazes de entender. Já testemunhei pessoas sustentarem comunicações no WhatsApp por dias usando apenas figurinhas. Nem uma mísera letrinha. Não foi para isso que inventaram o português, divago.

Entendem onde quero chegar? Existem inúmeras formas de comunicação e não existe uma ideal para todas as ocasiões. As formas, diferenciam-se entre si de diversas maneiras e se sobressaem dependendo da circunstância.

Figura 11.3: Código para "bola rápida, semi-devagar, retilínea e uniformemente variada.".

Telefone sem fio

Existe uma imortal brincadeira infantil que nos ajuda a entender de forma lúdica a importância dos códigos na comunicação.

Telefone sem fio é um jogo no qual uma pessoa (*remetente*) fala (*código*) uma frase (*mensagem*) para o participante ao seu lado (*destinatário*), de maneira que os demais não escutem. Quem ouviu tenta então repeti-la fielmente — se você jogar com pessoas honestas, o que parece nunca ter sido meu caso — para o próximo

participante, e assim por diante até chegar ao último, que deve falar a frase em voz alta.

Figura 11.4: Só digo uma coisa: eu digo é nada.

Uma das regras do jogo é que o ouvinte da vez não pode solicitar que a frase seja repetida.

Os ruídos de código são acumulados sucessivamente e é comum a frase tornar-se cada vez mais deturpada à medida que é repassada, eventualmente ficando totalmente diferente ao ouvinte final.

Quanto mais jogadores participam, mais distorcida fica a mensagem. A ponto de, nas minhas jogatinas, a mensagem original "De quem é esse jegue?" virar *"voulevu cuchê avemuá?"*. A brincadeira é simples e a diversão é quase garantida.

11.1 Protocolos de comunicação

Softwares estão incrementalmente dependentes de outros softwares para providenciar suas funcionalidades. Assim eles consomem dados de muitas fontes de informação, como apps, websites, APIs, arquivos, banco de dados, filas e muitas outras.

Esses componentes atuam como receptores e/ou transmissores e a mensagem é composta pelos dados que desejamos transitar. Invariavelmente, é necessário integrá-los de alguma forma.

Portanto, um mecanismo de integração se faz necessário para que diferentes sistemas dialoguem entre si.

No contexto de sistemas/processos/softwares, chamamos o código dos modelos da teoria da comunicação de protocolos. Os protocolos são um dos principais ingredientes para a comunicação e muitas vezes pode ser entendido como o próprio código.

Um protocolo de comunicação é um conjunto de regras que permite que duas ou mais entidades de um sistema troquem informações. O protocolo define as regras, a sintaxe, a semântica e a sincronização de dados. Eles são implementados por hardware, software ou uma combinação de ambos e devem ser acordados pelas partes envolvidas.

Atributos dos protocolos de comunicação

Existem grandes quantidades de protocolos e nada impede que você crie o seu próprio. Bem como os códigos da comunicação humana anteriormente exemplificados, os protocolos diferenciam-se entre si por diversos atributos. Tem para todo gosto e fetiche. Alguns desses atributos são:

- **Direção:** é possível que os comunicantes enviem mensagens uns aos outros ou essa habilidade é restrita apenas a um dos participantes? Em uma palestra, geralmente, apenas o palestrante consegue enviar uma mensagem.

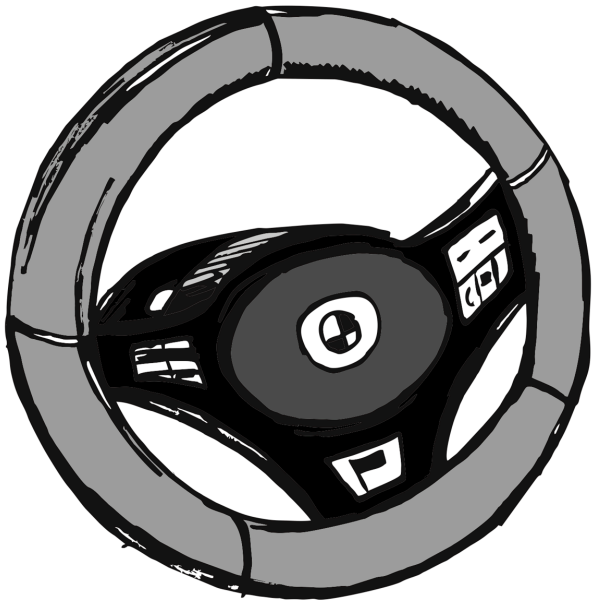

Figura 11.5: Direção.

- **Confiabilidade:** existe um grau de confiança com que as mensagens chegarão aos ouvintes? A ordem das palavras será mantida? Substitua esse ponto pela sua operadora telefônica, ou por crianças que trapaceiam no telefone sem fio, e compreenderá o que isso significa.

- **Peso:** quantos *bytes* são adicionados para enviar a mensagem?

  No alfabeto aeronáutico, para evitar ambiguidades, cada letra se torna uma palavra. A letra "A" é representada pela palavra "alpha" e a letra "B", por "bravo". Funciona, mas a mensagem se torna múltiplas vezes maior do que a original. A palavra **abacate**, por exemplo, com suas humildes 7 letras, é traduzida para *Alfa Bravo Alfa Charlie Alfa Tango Echo*, com 39 caracteres. Um aumento expressivo de, se eu não errei os cálculos, 457%, inviabilizando seu uso para mensagens muito grandes.

Figura 11.6: Peso.

- **Facilidade:** O que é mais fácil: aprender a língua do pê ou japonês em braile?
- **Sincronia:** ao enviar a mensagem, o remetente precisa ficar plantado esperando, como uma chamada no telefone fixo, ou pode receber uma resposta posteriormente, como um e-mail?
- **Quantidade de destinatários:** é possível enviar a mensagem para múltiplos ouvintes ao mesmo tempo, como em um grupo de WhatsApp. Essa característica permite a eternidade do folclórico gemidão do WhatsApp.

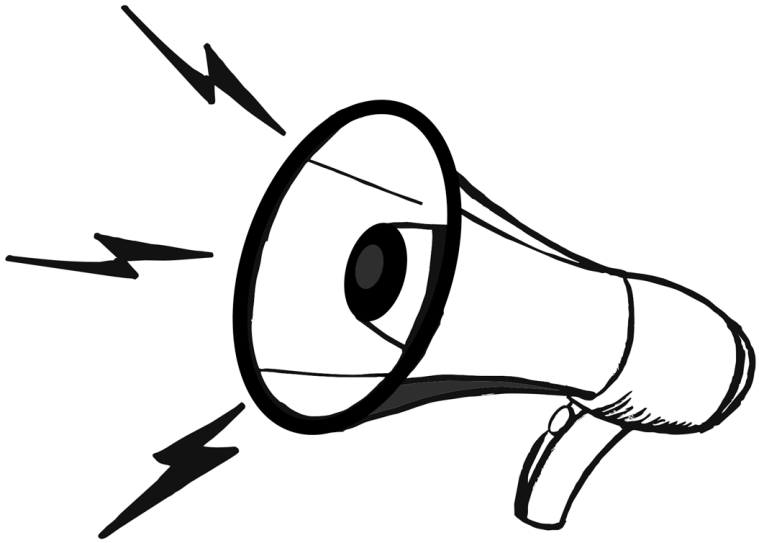

Figura 11.7: Protocolo unidirecional com muitos destinatários.

- **Outros:** acreditem, existem inúmeras outras características. Mas eu sou filho de deus – contrariando boatos infundados dos *haters* – e vou poupar meus dedos e sua leitura.

Sincronia

Destaco aqui o valor da sincronia. Ninguém merece ficar bloqueado enquanto não recebe uma resposta. Isso causa incômodo em alguns usuários do WhatsApp, mas, se é necessária uma resposta imediata, recomendo a utilização de outro protocolo bem conhecido: ligação telefônica – mãe, esse trecho é para você.

Eu, quando sou destinatário de uma mensagem, possuo outras prioridades – leia: *candy crush* e Netflix – e muitas vezes prefiro não responder imediatamente. Essa pode ser considerada uma ação egoísta e, na maior parte do tempo, é mesmo. Não há vergonha em admitir isso. Tal qual nos microsserviços, existem momentos em que isso é absolutamente a coisa correta a se fazer.

Comunicações assíncronas são compulsórias nos sistemas reativos. O que não significa que você precise usar um protocolo assíncrono. Quer dizer que a comunicação deve ser feita pela propagação de

dados assincronamente, não dependendo de outros como parte da operação inicial de solicitação.

O destinatário da mensagem a processa somente quando disponível, evitando sobrecargas e congestionamentos desnecessários.

Aguardar a resposta de consulta para poder fornecer um resultado de requisição, além de *démodé*, deteriora a resiliência. Fazer com que o remetente aguarde, impaciente, enquanto um intermediário faz a consulta é coisa do passado, essa não é a realidade dos *millennials*.

Figura 11.8: Sistemas reativos, não confundir com sistemas retroativos.

Além disso, como vimos no capítulo *Dificuldades da reatividade*, ao criar cadeias síncronas de requisição/resposta, o desempenho geral e a responsividade são diretamente afetados quando um dos serviços nessa cadeia não é executado corretamente.

Contudo, a comunicação assíncrona é naturalmente mais complexa e requer maior precaução. Afinal, não há, necessariamente, uma resposta para cada requisição. Como veremos na parte *Elasticidade*,

se não implementada com a devida atenção, pode acarretar problemas similares a *loops* infinitos e condições de corrida.

11.2 Fluxos poliglotas

Arquiteturas de microsserviços são compostas por múltiplos componentes e, não raramente, múltiplas gerações de tecnologias coexistem pacificamente fornecendo um produto viável.

Considerando também que em um conjunto de sistemas há várias trocas de mensagens com características diferentes, é razoável admitir a coexistência de múltiplos protocolos dentro de um mesmo ecossistema de serviços, o que denomino de **fluxos poliglotas**.

> **FLUXO POLIGLOTA**
>
> Ao transitar mensagens entre sistemas distribuídos, a escolha do protocolo de comunicação deve ser baseada nas aplicações envolvidas, na atual conjuntura do ecossistema e no que os dados representam.

Diferentes fluxos podem ser tratados com diferentes tecnologias. Isso significa permitir o uso da melhor ferramenta para o melhor caso de uso. A **diversidade tecnológica** é um dos principais pontos fortes dos microsserviços e é o mesmo argumento que sustenta a programação e a persistência poliglota, que falam sobre a viabilidade de um mix na escolha da linguagem de programação e tecnologia de persistência, respectivamente.

Ruídos na comunicação de sistemas

Assim como na brincadeira de telefone sem fio, falhas de comunicação se acumulam ao passar a informação sucessivamente,

potencializando ruídos, como *delays*, mal-entendidos e o desgaste de desempenho em cada retransmissão.

Cada sistema processa a mensagem antes de repassá-la adiante. Portanto, assim como na brincadeira, a mensagem é reinterpretada a bel-prazer antes de ser retransmitida e já sabemos no que isso pode resultar.

Além disso, a própria escrita do código possui defeitos inerentes e, se encararmos um sistema como um ninho de bugs em potencial, o que é uma realidade, veremos que cada reinterpretação deteriora a mensagem original em algum nível ao adicionar ruídos.

Portanto, o ideal é tentar minimizar as comunicações entre os componentes internos da arquitetura a fim de evitar acúmulos de ruído na comunicação. Apesar de provocar gostosas gargalhadas na brincadeira do telefone sem fio, os ruídos podem custar dolorosas quantias na vida real.

11.3 Conclusão

A escolha do protocolo de comunicação deve ser baseada em vários aspectos que variam dependendo do sistema e do contexto.

Às vezes, há premissas que excluem algumas possibilidades e limitam as opções, tais como escassez de luz, no mundo real, e latência, sincronia e números de atores envolvidos no mundo da computação.

Protocolos mais leves são preferíveis em ambientes com restrição de recursos computacionais. Do mesmo modo, o verbal é mais leve do que o aeronáutico quando possível. Em um contexto com restrições mais brandas, a escolha de um protocolo mais robusto e com grau de confiança maior pode ser mais apropriada. Em outras ocasiões,

quando é necessário consumir recursos de terceiros, por exemplo, resta a você aceitar e se conformar com as condições.

É por isso que, apesar de desejável, não é viável que o código verbal seja aplicado para todas as comunicações existentes. Simplesmente beira o impossível utilizá-lo em longas distâncias, com a boca cheia de farofa – isso é discutível –, com alguém que não fala o mesmo idioma e em muitos outros cenários.

É válido lembrar, todavia, que adicionar mais uma tecnologia à pilha tecnológica do projeto tem um preço, e que o preço não é uma pechincha. Ser a tecnologia mais adequada pode não ser suficiente para justificar a adoção no longo prazo. Adicionar tecnologias pode ser fácil, conviver com elas pode não ter a mesma facilidade.

Ainda assim, é um preço que pagamos diariamente para habilitar uma conversa humana a longuíssimas distâncias. Vários códigos atuam em conjunto e em harmonia. Códigos analógicos, digitais, luminosos, de rádio e tantos outros atuam em sequência para que, finalmente, a humanidade consiga tecer conspirações para descreditar o avanço científico dos últimos séculos.

CAPÍTULO 12
Por que (de novo) REST/http?

Há uma sensação de que microsserviços e APIs (*application programming interface*, em português, interface de programação de aplicações) RESTful em conjunto com o protocolo http passeiam sempre de mãos dadas.

De alguma forma, microsserviços e http viraram, praticamente, sinônimos. Independentemente do problema e do contexto, percebe-se a mesma combinação sendo escolhida como o paradigma de troca de mensagens padrão em uma arquitetura de microsserviços: APIs RESTful, *endpoints* http e mensagens JSON sendo propagadas como pombos sem asa.

Ao que parece, o conjunto REST e http é a única alternativa e a solução universal para a troca de mensagens entre serviços.

Figura 12.1: Outra vez café? Por isso ele só te traz flores!

Meus tesouros, antes de enveredarmos mais nesse estudo, adianto uma categórica negação. Http não é a solução universal. O protocolo

http é, assim como outras dezenas, uma das opções. Muitos pontos devem ser equacionados antes de se escolher a forma de comunicação entre os serviços envolvidos, considerando, como dito anteriormente, a possibilidade de que mais de um protocolo seja utilizado no mesmo fluxo.

12.1 Dando número aos bois

Por meio de uma brevíssima pesquisa nos pacotes disponíveis no repositório *npm*, levantei os protocolos de comunicação mais utilizados, suas respectivas bibliotecas mais populares e a quantidade de downloads feitos nos últimos cinco anos (dia 1 de janeiro de 2016 até o dia 31 de dezembro de 2020).

Para os que não estão familiarizados, *npm* (*node package manager*, gerenciador de pacotes do node) é o gerenciador padrão para o ambiente de tempo de execução Node.js. Entre outras coisas, ele é um repositório com mais de 1 milhão de pacotes disponíveis gratuitamente e quase 1 trilhão de downloads agregados por ano.

Sei que *npm* não representa a melhor fonte de verdade, que existem muitas outras finalidades para as bibliotecas em questão e reforço que a pesquisa foi realmente bem breve, mas é suficiente para ilustrar meu ponto.

Os protocolos pesquisados e suas bibliotecas correspondentes mais populares foram:

- **http**, com a biblioteca *express*.
- **amqp**, com a biblioteca *amqplib*.
- **mqtt**, com a biblioteca *mqtt*.
- **kafka**, com a biblioteca *kafka-node*.
- **stomp**, com a biblioteca *stompjs*.

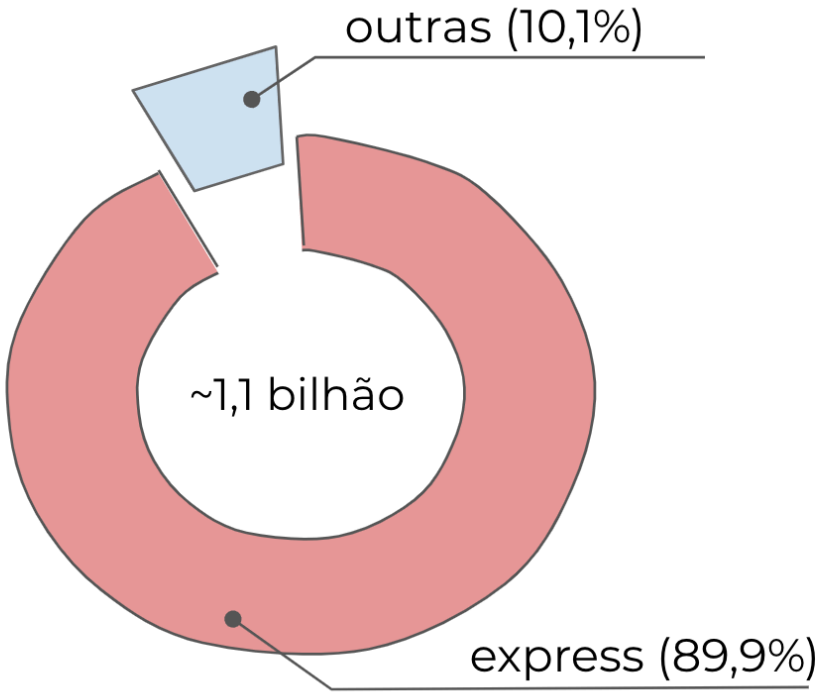

Figura 12.2: Http contra o resto.

O resultado não é nada surpreendente. Enquanto quase 1 bilhão de downloads foram feitos da biblioteca *express*, que disponibiliza um servidor http, as outras bibliotecas, somadas, possuem apenas 10% dessa fatia. A diferença é significativa e chama muita atenção.

Um ponto que vale lembrar é que a biblioteca *express* representa apenas metade de uma solução http, isto é, ela fornece implementações apenas para o servidor. A outra metade, a que representa o cliente, possui uma implementação padrão em JavaScript e, por isso, não foi contabilizada na pesquisa.

A questão é: em 90% dos contextos, http é a solução mais adequada?

Quando se analisa a evolução das curvas, apesar de se notar um crescimento recente nas outras bibliotecas, os valores absolutos ainda são irrisórios se comparados aos números de downloads da que representa o http.

Contudo, gosto de destacar que, enquanto em 2016 o download da biblioteca *express* representava 95% do total, esse montante diminui ano após ano, atingindo a marca de 89% em 2020.

Figura 12.3: Http com o express se distanciando ainda mais.

De onde vem essa discrepância?

Não é uma resposta fácil. É provável que seja uma combinação de diversos fatores. Provavelmente, o uso massivo se dê por sua facilidade ou pela infinidade de opções disponíveis no mercado. Outra explicação que possuo é que tutoriais amplamente divulgados na internet utilizam esse protocolo.

Ainda, aliado a esses fatos, atribuo uma parcela dessa responsabilidade ao texto onde Martin Fowler fala de microsserviços, em https://martinfowler.com/articles/microservices.html.

> *[...] se comunicando com mecanismos leves, frequentemente uma API com recursos http [...]* - Martin Fowler.

Apesar de ser uma afirmação vaga e de não soar como uma recomendação, acredito que o fato de o protocolo ter sido explicitamente mencionado por uma figura da envergadura do Martin Fowler pese e favoreça sua escolha.

O que é dito, ressalto, é que a comunicação ocorre por mecanismos leves, frequentemente http. Não há nenhuma exigência ou recomendação.

Observo também, empiricamente, uma adoção padrão dessa combinação pelo simples desconhecimento de outras alternativas.

12.2 Por que http?

Ao usar APIs RESTful baseadas no protocolo http, muitos benefícios são encontrados:

1. http é simples e familiar.
2. Suporta comunicação request/response diretamente.
3. Grande quantidade de bibliotecas disponíveis.
4. Não requer um sistema mediador, como *brokers*, o que simplifica a arquitetura do sistema.
5. Inúmeras ferramentas para testar a *API*, como *curl*, *browsers* comuns, *postman*, *insomnia*, *enqueuer* etc.

12.3 Por que não http?

Os benefícios enumerados não são suficientes para assumir http como a opção padrão e isso, assim como toda a franquia Velozes e Furiosos, precisa acabar urgentemente. Existem argumentos excelentes (ou terríveis, dependendo do ponto de vista) contra a adoção imediata do http:

1. Não há alternativas ao estilo *request/response* de interação.
2. Inexistência de um intermediário para armazenar as mensagens. Logo, cliente e servidor devem estar em execução durante a troca de mensagens.
3. O cliente deve conhecer o endereço de cada instância do servidor, o que prejudica a **transparência de localização**, um problema que não é trivial. Mecanismos de *service discovery* podem auxiliar a contornar essa dificuldade.
4. A natureza síncrona imprime um forte acoplamento entre o cliente e o servidor.
5. Requer que todas as comunicações sejam iniciadas pelo cliente.
6. Dificuldade da decomposição do servidor em serviços derivados quando necessário.
7. Exige a combinação de múltiplas requisições http para a composição de informações complexas.
8. URLs confusas para filtrar campos por meio de *query parameters*.
9. *Overfetching*, desperdício provocado quando dados além do necessário são enviados na mensagem.
10. A eterna confusão entre o *PATCH* e o *PUT*.
11. Por último, mas não menos importante: adotar as boas práticas RESTful para criação de *endpoints* é o ó do borogodó.

O fato é que o protocolo http não foi criado com essa intenção. Como o próprio nome sugere, o objetivo inicial era a transferência de hipermídia e objetos distribuídos e colaborativos, ou seja, texto estruturado que utiliza ligações lógicas (*hiperlinks*) entre os pontos.

Eu sei o que você deve estar pensando: "e daí que não foi criado com esse propósito?". De fato, a Coca-Cola também não foi criada com a intenção de ser refrigerante e deu no que deu.

Realmente, por si só, isso não configura um problema, mas vira um quando percebemos que o sabor do protocolo http não é tão agradável quanto o da Coca-Cola. Por exemplo, é muito amargo estar limitado apenas aos verbos http (*GET*, *POST*, *PATCH*, *PUT*, *DELETE* etc.) e aos códigos de status (200, 404, 500 etc.) para nos expressar.

Qual verbo você usaria para fazer uma validação de uma mensagem? Um *GET*, que não aceita payload, um *POST*, que não cria recurso? O código 404 significa que o recurso não foi encontrado ou que o *endpoint* não existe? É uma condição temporária ou permanente? Não vou nem comentar sobre os casos em que um código de status 200 é retornado junto com um erro no corpo da resposta.

Vou além, desafio você a abrir qualquer projeto que possua e encontrar métodos em suas interfaces que possuam apenas os verbos da especificação http.

A mensagem é clara, os verbos disponíveis são insuficientes e diminuem nosso vocabulário. Em um mundo onde dar nomes coesos aos identificadores no código é uma das tarefas mais árduas (segundo o já mencionado artigo de Martin Fowler), não podemos nos limitar ainda mais e abrir mão de um vocabulário extenso.

Ainda não está convencido? Além dos argumentos citados anteriormente, há uma extensa lista de artigos que estimulam uma reflexão prévia e a especulação de outros mecanismos de trocas de mensagens em vez da adoção imediata de http:

1. Esqueça http com microsserviços

   (https://bergie.iki.fi/blog/forget-http-microservices/)

2. Os dados não estão mais em REST

   (https://blog.ably.io/data-is-no-longer-at-rest-2cae0b1c5a81)

3. REST é o novo SOAP

(https://medium.freecodecamp.org/rest-is-the-new-soap-97ff6c09896d)

4. REST é a melhor escolha em uma arquitetura de microsserviços

   (https://capgemini.github.io/architecture/is-rest-best-microservices/)

5. Como REST síncrono transforma microsserviços de volta em monólitos

   (https://thenewstack.io/synchronous-rest-turns-microservices-back-monoliths/)

Figura 12.4: Dê um descanso ao REST.

12.4 Conclusão

É uma pena que o http seja amplamente considerado como o protocolo de comunicação de microsserviço padrão.

De modo geral, http, como todo protocolo síncrono, é desencorajado por promover o acoplamento, fragilizar a autonomia individual e a resiliência de toda a solução. Consequentemente, essa escolha desrespeita o que é pregado pelos microsserviços e os sistemas reativos.

Figura 12.5: Sistemas reativos, não confundir com sistemas respeitativos.

O paradigma REST + http como mecanismo de troca de mensagens traz consigo muitas desvantagens e, por conta disso, se traduz como uma péssima escolha padrão para comunicação entre microsserviços. Não que a combinação nunca deva ser usada. É possível que seja mesmo a solução mais adequada em muitos casos. O que se defende é que se faça um levantamento minucioso de todas as variáveis do problema em questão antes de escolher o protocolo mais apropriado.

A natureza distribuída dos aplicativos em nuvem requer uma infraestrutura de mensagens que conecte os componentes e serviços, idealmente de uma forma fracamente acoplada para maximizar a escalabilidade. O sistema de mensagens assíncronas é

amplamente usado e oferece muitos benefícios, mas também traz desafios como ordenação de mensagens, gerenciamento de mensagens suspeitas, idempotência e muito mais.

Parte 5: Elasticidade

Capítulo 13
Distribuição elástica

Anteriormente, vimos que a elasticidade é a responsividade sob demandas extremas. É a capacidade de automaticamente ser escalável se adaptando ao necessário. A escalabilidade envolve a distribuição do sistema. Não que não seja possível escalar sistemas não distribuídos, mas isso pode gerar ineficiência e custo adicional, uma vez que módulos não necessários serão escalados juntos.

Como veremos, a aliança entre a escalabilidade e a distribuição dos sistemas inevitavelmente produzirá inconsistência de dados.

13.1 Química dos dados

Como afirma Martin Kleppmann em *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*:

> *Os dados estão no centro dos principais desafios de design nos sistemas de hoje.* - Martin Kleppmann.

Dados representam a matéria-prima pelos quais organizações tomam decisões com o intuito de continuarem vivas e competindo em ambientes de negócio cada vez mais acirrados. Toda aplicação se baseia nos dados e o sucesso ou falha de qualquer negócio depende da eficiência da sua gerência de dados.

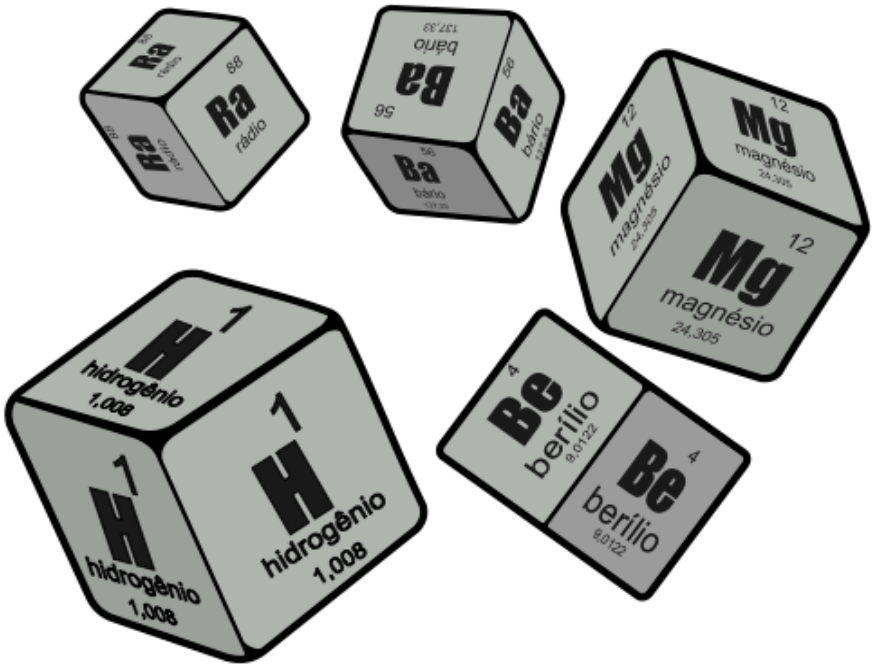

Figura 13.1: Dados químicos.

Em um SGBD (sistema gerenciador de banco de dados), uma transação é a unidade lógica de trabalho executada sobre... um banco de dados. Tratado de maneira coerente e confiável, independente de outras transações, sendo às vezes composto de várias operações, qualquer cálculo lógico feito em um banco de dados é denominado de transação.

Um exemplo, provavelmente o mais clássico do universo, é a transferência de uma conta bancária para outra. A transação completa requer a subtração do valor a ser transferido de uma conta e a adição da mesma quantia à outra. Neste exemplo, não é uma boa ideia que apenas uma operação, a adição ou a subtração, seja realizada com sucesso, embora fosse divertido se, ao transferir, nenhum dinheiro fosse debitado da minha conta. Assim, só existem duas possibilidades admissíveis: 1) as duas operações são bem-sucedidas, 2) nenhuma é.

A essa característica de exigir a completude de todas as operações, ou a total inexistência delas, dá-se o nome de **atomicidade**. Discorro um pouco sobre essa propriedade e algumas outras logo abaixo.

13.2 Transações ACID

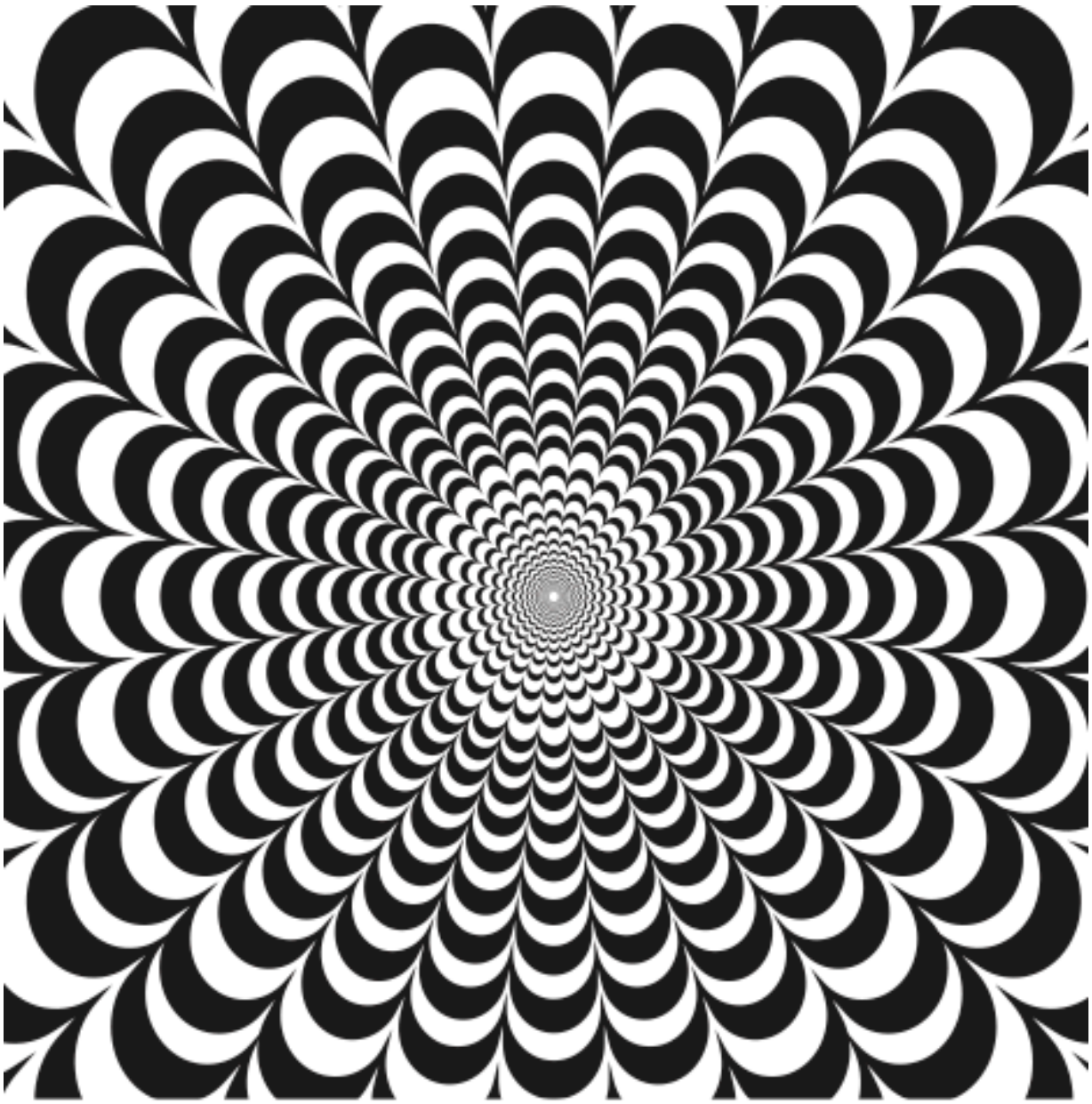

Figura 13.2: Ácido. Use em pequenas doses.

Em um banco de dados relacional, existe um conjunto de propriedades que garante que as transações sejam processadas de maneira confiável.

É comum se referir a essas propriedades de transações de bancos de dados usando o termo ACID. O acrônimo, embora superficialmente definido, é bastante popular na indústria desde,

pelo menos, a década de 1980 e define as transações como **atômicas**, **consistentes**, **isoladas** e **duráveis**.

- **Atomicidade:** todas as ações que compõem a unidade de trabalho da transação devem ser concluídas com sucesso para que ela seja efetivada. Se durante a transação qualquer ação que constitui unidade de trabalho falhar, a transação inteira deve ser desfeita (operação conhecida como *rollback*). Quando todas as ações são efetuadas com sucesso, a transação pode ser efetivada e persistida (no inglês, *commit*).
- **Consistência:** todas as regras e restrições definidas no banco de dados devem ser obedecidas. Relacionamentos por chaves estrangeiras, verificação de valores para campos restritos ou únicos devem ser obedecidos para que uma transação possa ser completada com sucesso.
- **Isolamento:** cada transação funciona completamente à parte de outras transações. Nenhuma outra transação, operando no mesmo sistema, pode interferir no funcionamento da transação corrente. Transações não podem visualizar os resultados parciais de outras.
- **Durabilidade:** os resultados de uma transação são permanentes e podem ser desfeitos somente por uma transação subsequente. Erros por falha de hardware não são aceitáveis.

Há uma relação estrita entre as propriedades ACID entre si. Por exemplo, o isolamento depende da atomicidade para reverter as alterações no caso de falhas de isolamento.

A consistência também depende da reversão no caso de uma violação de consistência por uma transação ilegal. Finalmente, a atomicidade depende da durabilidade para garantir seu funcionamento mesmo em face de falhas externas.

Essas propriedades reinaram soberanas no mundo durante décadas sem muita resistência, porém sua natureza restritiva limita sua aplicação e, por vezes, não são compatíveis com os softwares modernos.

Figura 13.3: Sistemas reativos, não confundir com sistemas restritivos.

Dependendo da situação, é perfeitamente aceitável abrir mão de uma delas visando a elasticidade de um sistema reativo. Veja, a durabilidade de um cache pode não ser tão importante quanto uma busca mais eficiente, ou o custo do acesso ao disco rígido pode ser mais caro do que o custo da inconsistência.

Ao se permitir afrouxar a rigidez dessas características, um espectro inteiro de transações de dados se abre, no qual, em um extremo, encontramos as transações ACID e, no resto, encontramos o que é chamado de transações BASE.

Antes de falar sobre a definição dessas transações, permitam-me falar um pouco sobre química.

Química avançada

Aproveitando a ocasião, destaco que a expressão "atomicidade" também é oriunda da química. Lá pelas tantas de algum século aí,

os cientistas usaram a palavra "átomo" pela primeira vez para descrever o que acreditavam ser o menor pedaço de um elemento.

Eles assim o denominaram, porque o significado, do grego, é "indivisível". Formado a partir de ἀ (a, "não") e τέμνω (temnō, "cortar"). Posteriormente, foi descoberto que os átomos podem ser divididos, mas já era muito tarde para mudar um termo que circulava no Twitter dos gregos.

No atômico do ACID, assim como no sentido original da química, as ações que compõem a transação são indivisíveis. Tudo ou nada, oito ou oitenta, sem meio termo, independentemente do que os químicos argumentarem.

Eles que venham programar, caso persistam na teimosia.

> **Reatividade química**
>
> A reatividade também é um conceito químico. Na química, ela define o ímpeto pelo qual uma substância sofre uma reação química por si mesma ou com outros materiais, com uma liberação total de energia. Só não acho que seja muito útil para a computação – ainda.

Assim como "átomo", os termos "ácidos" e "básicos" também são amplamente utilizados na química e reagem entre si em uma transação chamada de **neutralização**.

Por que criei esta seção? Primeiro, para dizer que lembro de algumas aulas de química. Segundo, porque pretendo valorizar a criatividade e o empenho da galera de desenvolvimento de software. Foram lá na química achar umas propriedades e fizeram contorcionismo para fazer delas conceitos computacionais – embora sua precisão e utilidade sejam discutíveis.

Fica aqui registrado meu respeito pelas pessoas que fizeram isso.

13.3 Transações BASE

Figura 13.4: Jogo de bases.

O acrônimo BASE foi definido por Eric Brewer – a quem destinei meu respeito no parágrafo anterior – e é usado para descrever as propriedades de alguns bancos de dados, geralmente bancos NoSQL.

Em tempo, NoSQL é um tipo de banco de dados desenvolvido para providenciar uma resposta onde os bancos relacionais tradicionais pecavam. Principalmente, mas não se limitando, à manipulação,

variedade e velocidade das informações em volumes massivos de dados distribuídos.

A sigla talvez seja ainda mais superficial do que ACID, pois tem um significado vago e abrangente. Ainda assim não posso deixar de explicar esse mosaico em forma de sigla: basicamente disponível, estado *soft* e consistência eventual (do original, em inglês, ***B****asically* ***A****vailable*, ***S****oft state*, ***E****ventual consistency*).

- **Basicamente disponível:** foco na disponibilidade de dados mesmo diante de múltiplas falhas por meio de uma abordagem de gerenciamento distribuído.
- **Estado soft:** sem uma atualização, os dados expirarão ou serão excluídos. O estado pode mudar com o tempo, mesmo sem solicitação. Ou seja, a consistência é um problema do desenvolvedor e não deve ser tratada pelo SGBD.
- **Consistência eventual:** em algum ponto do futuro, no entanto, sem garantias quanto ao momento, os dados convergirão para um estado consistente, conforme as alterações são propagadas para as réplicas. Enquanto isso, as consultas de dados podem não retornar dados atualizados ou podem redundar em leitura de dados não precisa, um problema conhecido como leitura obsoleta.

As propriedades BASE são bem mais frouxas do que as ACID e não há um mapeamento direto entre uma e outra – um argumento que provavelmente será contestado.

13.4 Balanceamento ACID-BASE

Figura 13.5: Espectro de transações de dados.

Essa propriedade se trata de um espectro e não de uma escolha binária. Na química, o limão (pH 2,3) e a laranja (pH 3,5), apesar de ácidos, possuem graus distintos de acidez.

Valores intermediários de grau de acidez também se aplicam nas transações de dados. Uma transação não é sempre ácida ou 100% básica. Valores intermediários são perfeitamente aceitáveis.

Perceba que, em um *e-commerce* – porque "exemplo" e "*e-commerce*" são quase sinônimos –, a diferença entre ter 42 ou 43 livros no estoque é, muitas vezes, irrisória. Logo, neste caso, podemos relaxar na consistência dessa operação.

Daqui poucos segundos, a quantidade de livros pode ser atualizada para 43. Ou mais, é possível que seja mais interessante para o negócio oferecer um desconto em outra compra, como compensação

por conta de uma venda indevida, do que manter a consistência perfeita e prejudicar a experiência de mais usuários.

Não existe uma resposta direta sobre se sua aplicação precisa de um modelo ACID ou BASE. Não é um cálculo trivial e muitas regras de negócio devem ser consideradas. Devs e arquitetos(as) de dados investem muitas horas ponderando o ponto ideal para a circunstância com base no que é mais importante para a situação e não apenas no que está na tabela periódica ou no modelo previamente adotado - cogitando, inclusive, múltiplas adoções para resolver o problema.

Por mais que abrir mão de um dos pontos das transações ACID possa parecer inicialmente doloroso e arriscado, reforça-se que a maioria dos bancos NoSQL o faz. Redis, por exemplo, desconsidera a durabilidade; Cassandra, a consistência, e o Riak, a atomicidade. No entanto, alguns outros, como MarkLogic, Aerospike e OrientDB tornaram essas propriedades centrais em seus projetos.

13.5 Conclusão

Banco de dados relacionais, geralmente ácidos, e não relacionais, geralmente não ácidos básicos, não são mutualmente exclusivos.

Como prega a persistência poliglota, muitos casos de uso aproveitam as maneiras como esses dois bancos de dados diferentes podem se complementar. Cada um traz seus próprios pontos fortes, tornando a soma maior do que cada parte. Os bancos relacionais costumam lidar com dados estruturados e padronizam como os elementos se relacionam entre si. Os bancos de dados não relacionais são populares quando a flexibilidade é uma preocupação.

Em um sistema monolítico, o SGBD garante o pH ácido ou básico das transações, mas como dosá-lo nos sistemas distribuídos? Não há, por padrão, um coordenador único de transações. Possuir todos

SGBDs ácidos não é suficiente para que haja a garantia de que todas as transações entre eles sejam ácidas também. O fato de um dos sistemas possuir uma base de dados com operações atômicas não significa que as transações por todo o ecossistema também sejam atômicas e o mesmo se aplica para as demais propriedades: consistência, isolamento e durabilidade.

Individualmente, para cada SGBD, os acrônimos ACID e BASE permanecem fazendo sentido e a analogia com a química é válida. No entanto, ao observarmos pela ótica da distribuição de dados, em uma visão do todo, o prisma ácido/básico de um único SGBD já não é tão relevante, fazendo com que a química não tenha vez, o que é lastimável, porque eu realmente gostei dos enxertos inspirados na química inorgânica.

Figura 13.6: Não rola química.

Assim que saímos da fronteira da instância de um serviço, entramos no oceano indomável do não determinismo, o mundo caótico dos sistemas distribuídos.

De certa forma, esse oceano guarda uma similaridade com os Correios do Brasil. Em ambos, as falhas ocorrem das maneiras mais espetaculares, a detecção de falhas é um jogo de adivinhação, e as mensagens são perdidas, reordenadas e distorcidas.

É chato. Vocês vão amar.

Capítulo 14
Jogo de dados

Nas glamorosas aplicações distribuídas, em vez de gerenciar todos os dados de todos os domínios com um único SGBD compartilhado, prática conhecida como *Data-Centric Principle* (princípio centrado em dados - DCP), é comum e fortemente recomendável encontrarmos ecossistemas que seguem a prática imposta pelo princípio *Single Repository Principle* (Princípio de Repositório Único - SRP).

Essa prática exige que cada microsserviço mantenha seu próprio banco de dados, proporcionando largura de banda, baixa latência e alta disponibilidade, e que nenhum serviço acesse a base de dados de outro serviço diretamente.

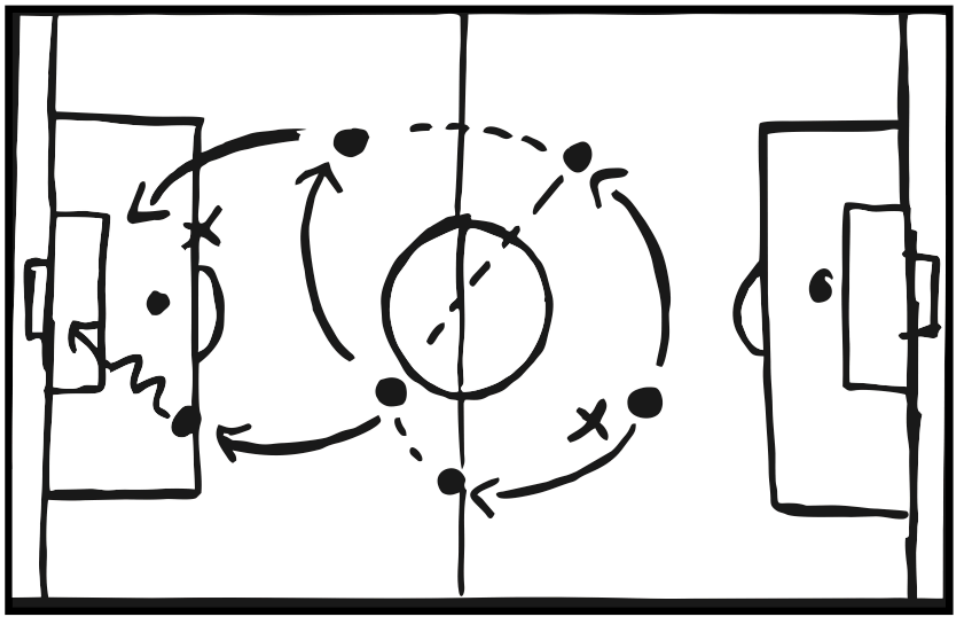

Figura 14.1: Jogo de dados.

Como você sabe, os desenvolvedores não resistem à tentação de burlar qualquer proteção para acessar o conteúdo diretamente, portanto convém criar obstáculos que reforcem a proteção e a modularidade. Um desses obstáculos consiste em distribuir cada base de dados em um servidor. Existem maneiras alternativas que mantêm o escopo de acesso restrito e separam os domínios, como tabelas privadas por serviço e *schemas* por serviço.

Uma plataforma *e-commerce*, como um exemplo para variar um pouco, lida com muitos dados distintos: endereços, estoques, preços, pagamentos, pedidos e outros. Cada tipo tem uma exigência específica: prioridades, desempenhos, valores, consistências etc.

Essa prática propicia a cada aplicação a oportunidade de atuar como uma camada de negócio antes da persistência, possibilitando o tratamento da informação com as regras do domínio.

Cito outras boas razões para adotar o Princípio de Repositório Único e criar bancos de dados individuais para cada serviço: melhor encapsulamento, disponibilidade individual, facilidade de administração, possibilidade de agendamentos periódicos personalizados, escala sob demanda, manutenção apropriada, recuperação de dados, otimização de desempenho, menos conversões e formatações de valores, capacidade de planejamento etc.

Algumas dessas vantagens são muito importantes, justificando, por si só, a adoção dessa estratégia. Outras nem tanto e só constam no parágrafo para fazer volume. Existem ainda mais, mas vou parar por aqui, porque já deu, né? Você já captou a ideia por trás dos meus argumentos.

Aliás, vou citar apenas mais uma: **persistência poliglota**.

Persistência poliglota

De modo similar ao fluxo poliglota e à programação poliglota – e, diferentemente do ex-técnico de futebol e eterno meme Joel

Santana –, o poliglotismo na persistência afirma que, para cada cenário, é melhor adotar o modelo SGBD mais apropriado, baseado nas necessidades e em como o dado é manipulado.

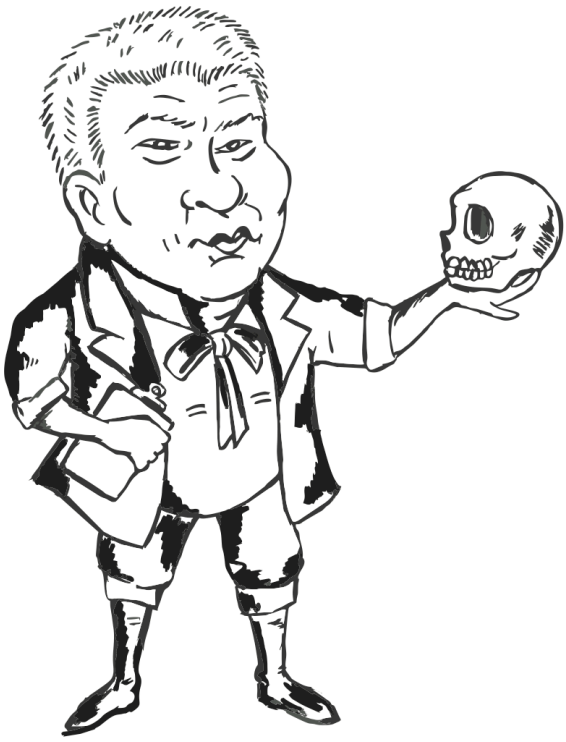

Figura 14.2: Tá de brinqueichon with me?

Adotar um modelo SGBD significa escolher a melhor ferramenta para determinado caso, mesmo que isso acarrete múltiplas tecnologias de persistência compartilhando um ecossistema.

14.1 Disputa de dados

Ao passo que SGBDs são sobre expor os dados e torná-los úteis, serviços são, paradoxalmente, sobre escondê-los para desacoplar. É uma guerra ferrenha, tipo um biscoito vs. bolacha, ou como um Brasil vs. Argentina, disputando pelo grau de exposição dos dados e gentilmente trocando sopapos.

No mundo dos sonhos, haveria unanimidade com a palavra "biscoito", os microsserviços seriam autossuficientes com suas informações e não haveria violência no clássico do futebol sul-americano.

Assim, transações entre serviços e batalhas pelos dados não seriam sequer necessárias, frustrando os amantes da pancadaria no futebol.

Mas o buraco é mais embaixo. Nos microsserviços, as regras de negócio possuem elementos baseados nos dados alheios, justificando cotoveladas e entradas perigosas, o que torna o encapsulamento de dados um quebra-pau violento, tal qual um Palmeiras vs. Corinthians na final do Paulistão de 1999. Aquele em que o chinelo canta solto após as embaixadinhas do Edílson Capetinha.

Figura 14.3: Edílson Capetinha, segundos antes da treta.

Como transações entre os serviços são inevitáveis, temos que nos preparar para todos os poréns que a comunicação entre sistemas traz: perdas, reordenamento de pacotes, complexidade, confiabilidade, latência...

Resta que nos contentemos e nos preparemos para o pior.

14.2 Transações frustradas

Assim como os chutes do Edílson Capetinha, as transações nem sempre são bem-sucedidas.

A internet é estocástica e partições de rede fazem parte do jogo de incerteza. Sistemas ficam indisponíveis e pacotes se perdem. Entretanto, os sistemas distribuídos, como um todo, não precisam padecer do mesmo mal.

É válido interpretar as partições de rede como as falhas inerentes ao ambiente distribuído sendo, portanto, termos perfeitamente intercambiáveis no contexto.

Felizmente, a replicação de dados e a redundância permitem a possibilidade de recuperação em momentos de partições de rede e o fornecimento de valores mesmo quando um sistema estiver indisponível ou diante de perdas de mensagens - o que habilita a tolerância a falhas e escancara uma possível falta de consistência. Isso reforça a reatividade pelo aspecto resiliente do sistema.

De outro modo, também por meio da replicação, poderíamos tratar as falhas de modo consistente, mas abriríamos mão da disponibilidade, fazendo com que nem todas as requisições tenham resposta.

Olhem que maravilha, a replicação permitiria que o Edílson Capetinha errasse vários chutes a gol consistentemente. Ocasião

apropriada para citar um jargão do renomado narrador brasileiro Milton Leite: "*Que beleza!*".

> *"O Messi, para jogar mais do que eu, tem que ser campeão mundial*.".
>
> *"Sou mais habilidoso do que o Cristiano Ronaldo*." – Edílson Capetinha, pentacampeão mundial de futebol.

No nível de abstração dos sistemas distribuídos, o pobre espectro ácido/básico, já limitado por natureza, é deliberadamente ignorado. Tanto por responder pouco sobre o ecossistema em que atua quanto por ignorar as propriedades dos fluxos de dados.

Por conta disso, os acrônimos ACID e BASE recebem auxílio de um acrônimo mais fashion e descolado: CAP – *consistency*, *availability* and *partition tolerance* (despretensiosamente traduzido do inglês: **consistência**, **disponibilidade** e **tolerância a partições**).

No capítulo anterior, perguntávamos: "qual o pH adequado para a solução?". Neste, perguntamos: "como obter a consistência, disponibilidade e a tolerância a partições?"

Se você tentou atingir esses três pontos simultaneamente, deve ter percebido que, assim como na vida, nunca é tarde demais para parar de tentar.

Com um amargor no peito, confirmo: essa maximização é tão inalcançável quanto o hexacampeonato da seleção brasileira de futebol.

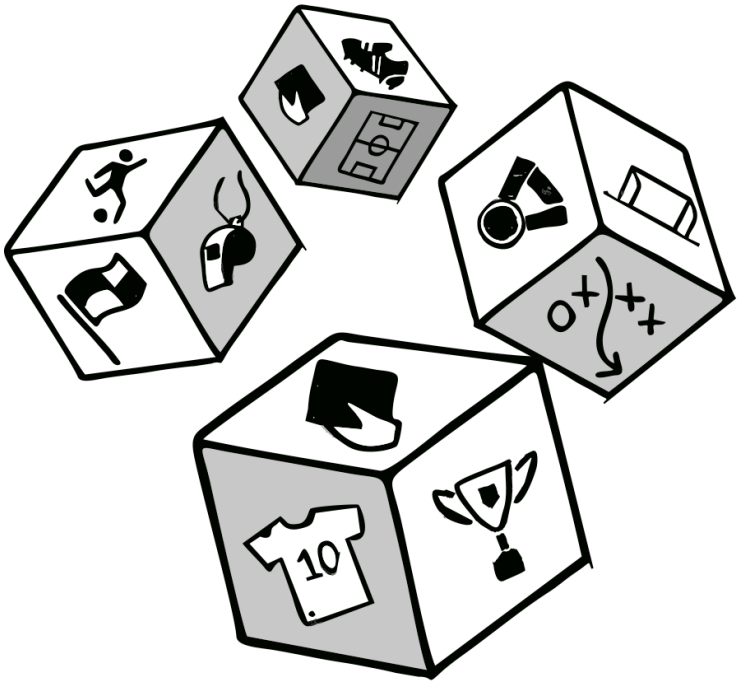

Figura 14.4: Dados de futebol.

Isso é precisamente – excetuando-se a parte do hexa – o que diz o **teorema CAP**.

14.3 Teorema CAP

O teorema declara, categórica e matematicamente, que, uma vez que existem contradições em termos entre os três itens, não é possível obtê-los simultaneamente. Simplificando:

> Consistência, disponibilidade e tolerância a falhas: escolha, no máximo, dois.

Em 2000, Eric Brewer, o mesmo que batizou o acrônimo BASE e a quem parabenizei no capítulo anterior, apresentou sua palestra à comunidade e introduziu ao mundo o teorema CAP.

Também conhecido como o teorema de Brewer, o teorema CAP relata a existência de três requisitos essenciais necessários para o sucesso de um projeto distribuído: **consistência**, **disponibilidade** e **tolerância a partições**:

- **Consistência (*consistency*):** todos os componentes veem os mesmos dados ao mesmo tempo.
- **Disponibilidade (*availability*):** as requisições sempre recebem uma resposta de falha ou de sucesso.
- **Tolerância a Partições (*partition tolerance*):** um determinado sistema distribuído continua a operar mesmo com falha na rede que conecta seus nós.

Essas são definições simples dos três aspectos do teorema. Não dá para ter todos simultaneamente. Dá para ter no máximo dois e no mínimo, nenhum. Pessoalmente, não vejo muita vantagem na última opção.

Assim como o espectro ACID-BASE, o teorema CAP apresenta uma escala de escolhas de suas três propriedades: existem vários níveis de consistência, vários níveis de disponibilidade e vários graus de tolerância a partições.

Existem inúmeros artigos disponíveis que discutem as implicações do teorema no mundo real e debatê-los aqui no livro vai além do meu contrato.

Porém, alerto que ignorar as implicações do teorema pode causar resultados catastróficos que incluem a possibilidade de não alcançar nem disponibilidade, nem consistência e nem tolerância a falhas. "Corram para as colinas" e "distribuam seus currículos" costumam ser escutados nessas ocasiões.

Apesar de sofrermos com os limites impostos pelo teorema CAP, também nos beneficiamos com ele. Ao permitirmos sistemas inconsistentes, desfrutamos da magia da escalabilidade, o que pode propiciar o aumento de disponibilidade.

Não que só seja possível escalar sistemas inconsistentes, o que pode ser verdade dependendo do modo de escalabilidade, mas escalar os sistemas sabidamente inconsistentes é uma tarefa simples.

14.4 Escalabilidade

A escalabilidade é um indicador importante na computação distribuída. Ela descreve a capacidade do sistema de ajustar dinamicamente seu próprio desempenho de computação, alterando os recursos de computação disponíveis e métodos de agendamento.

Nos sistemas distribuídos, há três dimensões de escalabilidade: x, y e z.

Figura 14.5: Vertical: mais para o mesmo. Horizontal: mais do mesmo.

Escalabilidade X

A escalabilidade x, também conhecida como escalabilidade horizontal, ou *scale out/in*, determina a quantidade de instâncias de uma aplicação.

Utilizar a escalabilidade x é como adicionar jogadores ao campo ou retirá-los de acordo com o nível do adversário e o placar do jogo. Parece uma boa ideia ter mais jogadores do que o adversário mesmo quando nos referimos ao Edílson Capetinha, não?

Escalabilidade Y

A alteração das cargas de trabalho alterando recursos de hardware, como memória e processadores, é conhecida como escalabilidade y, escalabilidade vertical, ou *scale up/down*.

Seria como oferecer ao Edílson Capetinha um coquetel de Red Bull, Gatorade e Biotônico Fontoura e transformá-lo no Pelé.

Escalabilidade Z

Além dos anteriores, a distribuição dos sistemas possibilita a otimização da performance através da repartição de dados, ou escalabilidade z – gosto de chamar de "escalabilidade transgonal", mas, por favor, não repitam isso em casa.

Com critérios predefinidos e uma função de partição, o serviço é dividido em partes independentes e distintas, e cada uma é responsável por cuidar de uma parcela dos dados do mesmo domínio. Uma entrada é gerenciada por uma, e somente uma, parte do serviço.

Seria como se, no futebol, tivéssemos um time só para atacar e outro time só para defender.

14.5 Escalabilidade dinâmica

Implantar um aplicativo para produção com uma configuração estática não é o ideal. Os padrões de tráfego mudam rapidamente, de acordo com o horário, dia da semana, dia do mês, promoções e vários outros motivos, e o aplicativo deve ser capaz de se adaptar.

Em outras palavras, o aplicativo deve ser **elástico**, que, como vimos, é um dos principais objetivos dos sistemas reativos.

Quando a demanda aumenta, o aplicativo deve ser escalado para permanecer responsivo, quando a demanda diminui, o aplicativo deve ser reduzido para não desperdiçar recursos. Não faz sentido usar um computador superpotente da NASA quando até um palmtop resolveria seus problemas.

A não ser que você seja parente do Bill Gates – o que é estatisticamente improvável – é bom atentar para o fato de que a escalabilidade envolve custos, geralmente monetários, e, por isso, a alocação em excesso não é uma boa ideia. Idealmente, instalam-se métricas nos serviços para que o redimensionamento ocorra em tempo de execução.

De modo análogo não é eficiente entrar com 15 Pelés em campo quando o adversário se trata do Íbis (PE) e meio Pelé seria suficiente – sem querer ofender o glorioso esquadrão pernambucano, internacionalmente conhecido como o pior time do mundo, honraria concedida após seus incansáveis fracassos dentro de campo.

A escalabilidade dinâmica ajusta a capacidade dos recursos em resposta às mudanças ativas na utilização dos recursos. A intenção é fornecer capacidade suficiente para manter a utilização no valor de destino especificado pela estratégia de dimensionamento.

Isso é semelhante à maneira como o termostato mantém a temperatura da sua casa. Você escolhe a temperatura e o termostato faz o resto.

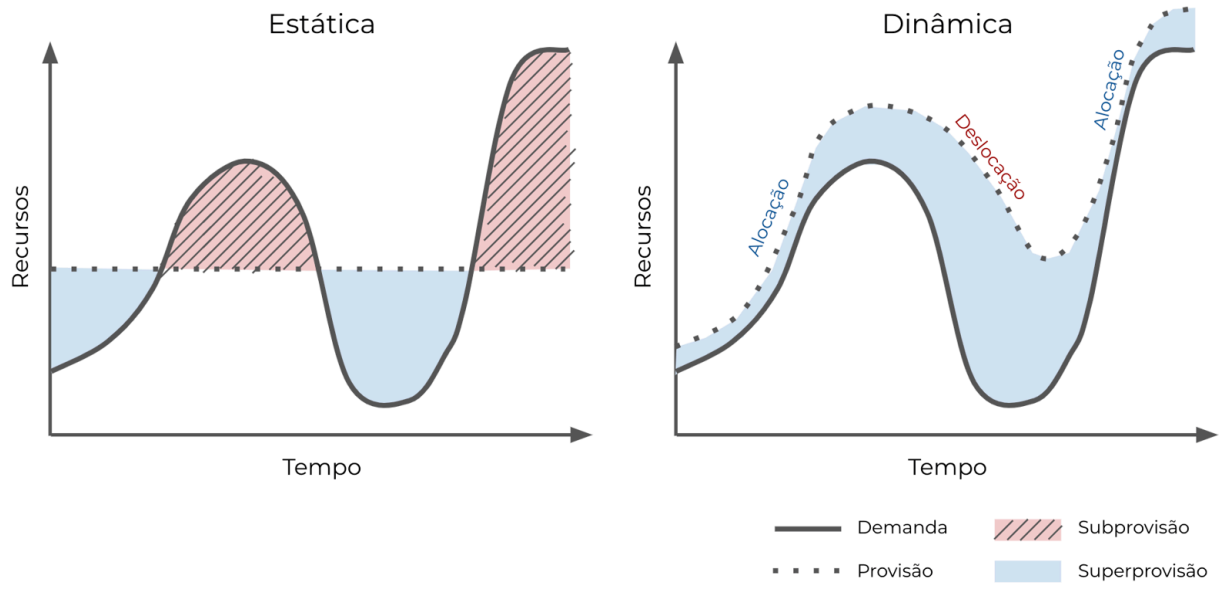

Figura 14.6: Escalabilidade dinâmica.

Escalar além do necessário resulta em **superprovisão**, ou seja, desperdício e custo, e fornecer menos do que o necessário acarreta **subprovisão**, o que quer dizer insuficiência de recursos, podendo gerar, no melhor dos casos, um gargalo de processamento e, em casos extremos, indisponibilidade.

14.6 Técnicas de escalabilidade

Baseado em valores parametrizáveis mínimos, máximos e desejados de métricas, como requisições por segundo, utilização de CPU ou memória, tempos de resposta, requisições por segundo, mensagens na fila e muitos outros, o **gerenciador de escalabilidade** coleta dados periodicamente e faz cálculos para saber se o valor atual está na **faixa de aceitabilidade**. Caso contrário, o gerenciador aloca ou desaloca recursos para manter a métrica dentro dos limites aceitáveis.

Orquestradores de contêineres, como o *Kubernetes* (https://kubernetes.io/) e *AWS - EC2 Auto Scaling* (autoescaladores de computação elástica em nuvem, em tradução livre, https://docs.aws.amazon.com/autoscaling/) atuam como gerenciadores de escalabilidade e fazem isso com uma mão nas costas.

As métricas que vão determinar a dimensão da elasticidade e o valor desejado variam e podem ser customizáveis. Técnicas de escalabilidade dinâmica envolvem diferentes domínios e algoritmos como abordagens baseadas em limites, análise de série temporal, teoria das filas, teoria de controle e aprendizado de máquina.

O aprendizado de máquina, por exemplo, pode analisar o histórico para prever regularmente qual será a necessidade no futuro.

Usando a previsão, a **escalabilidade preditiva** gera ações de escalonamento programadas para garantir que a capacidade do recurso esteja disponível antes que seu aplicativo precise dela, evitando subprovisionamento.

Você pode configurar sua estratégia de dimensionamento para manter a utilização média da CPU em 50%. Sua previsão indica que os picos de tráfego ocorrem todos os dias no almoço e na janta. Seu plano de dimensionamento agenda as ações futuras para garantir que sua aplicação esteja pronta para lidar com esse tráfego antecipadamente.

Isso ajuda a manter o desempenho constante, com o objetivo de sempre ter a capacidade de manter a utilização dos recursos o mais próximo de 50% sempre.

Na parte sobre *Resiliência*, o capítulo *Observabilidade de sistemas* é dedicado inteiramente às métricas e à observabilidade, no geral.

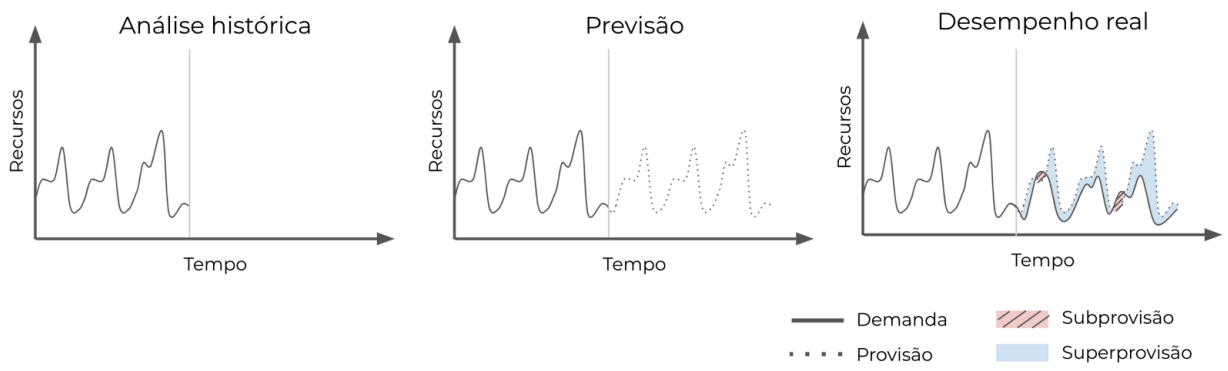

Figura 14.7: Escalabilidade preditiva.

O objetivo da escalabilidade dinâmica é ajustar a faixa de aceitabilidade de modo a evitar completamente a subprovisão enquanto minimiza a superprovisão. Como podem imaginar, não é uma tarefa simples.

As vantagens da **escalabilidade apropriada** incluem:

- **Custo:** o dimensionamento adequado de um software reduzirá o custo de alocação de recursos e manutenção.
- **Desempenho:** maior capacidade de processamento costuma acarretar melhor desempenho.
- **Distribuição de carga:** usando instâncias diferentes, podemos facilmente manter nossa carga balanceada, sem nenhuma instância sobrecarregada e nenhuma ociosa.

Limites da escalabilidade

Há um limite para os benefícios da escalabilidade. Primeiro, porque a curva do benefício possui comportamento assintótico quando a escalabilidade adotada tende ao infinito.

Segundo, porque a escalabilidade é limitada pela introdução de gargalos ou pontos de sincronização dentro do fluxo envolvido. Extrapolando ao limite e, mais uma vez, fazendo uma comparação com o esporte bretão, não há diferença de benefício se

escalássemos nosso time com um milhão de Pelés em vez de apenas 999.999. Essa nova adição sequer caberia no campo e se limitaria a atuar como espectador. Ou seja, há limites na escalabilidade. No caso do futebol, um desses limites é a dimensão do gramado.

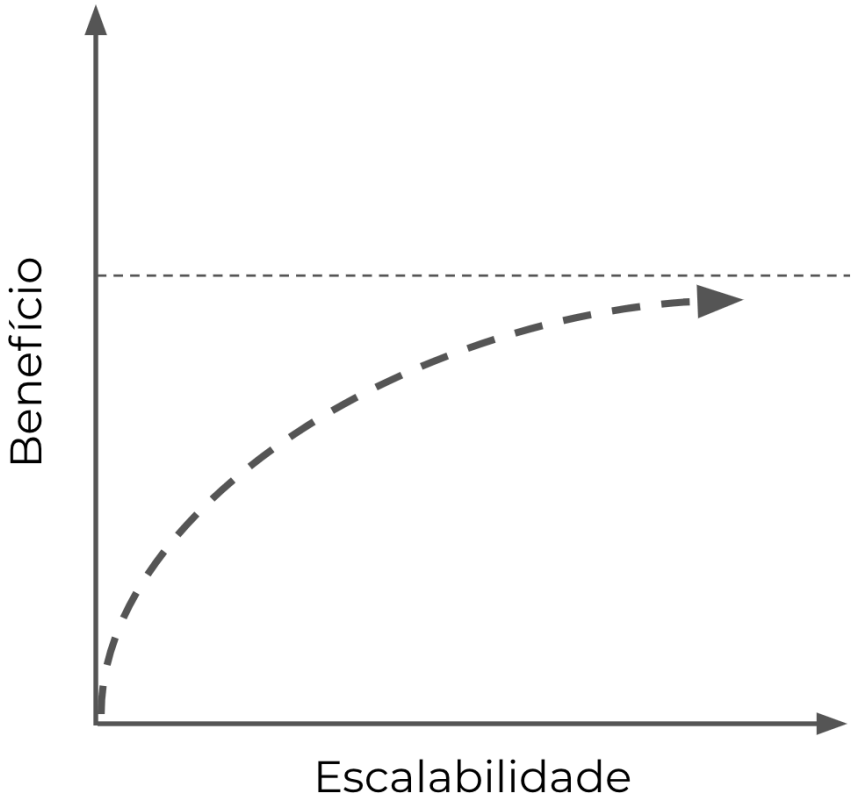

Figura 14.8: Escalabilidade x Benefício.

Foi o que argumentou Gene Amdahl quando concebeu a **Lei de Amdahl** no seu trabalho intitulado *The Logical Design of an Intermediate Speed Digital Computer*. A Lei de Amdahl resume a melhoria do desempenho de alguns processos, como um programa de computador. Ela considera que, para melhorar o desempenho, normalmente apenas uma parte de todo o processo é melhorada.

Isso mostra que a melhoria é limitada pela fração de tempo em que a parte melhorada é usada. Assim, entendemos que, não importa o grau de escalabilidade alcançado, o tempo de execução das demais

partes do fluxo é um limite superior prático no desempenho de sua contraparte escalada.

Com outras palavras, há um teto na melhoria do tempo de resposta de um serviço. Por exemplo, aumentar a configuração de hardware quando o tempo do hardware dedicado a essa tarefa representa uma fração insignificante do tempo total não representará um ganho real.

Pelo contrário, a alta escalabilidade é barrada por fatores, como 1) **contenção**, esperando que recursos compartilhados se tornem disponíveis, e 2) **coerência**, o atraso para que os dados se tornem consistentes.

A Lei de Amdahl explica que a contenção produz retornos decrescentes à medida que mais recursos são adicionados ao sistema.

A lei atua como uma ducha de água fria para quaisquer expectativas excessivamente otimistas que os desenvolvedores possam ter sobre os ganhos a serem obtidos com a escalabilidade.

Portanto, sem tentar diminuir o entusiasmo de toda a empresa, é crucial entender que o que se deve procurar é uma escalabilidade adequada e que isso é bastante desafiador.

14.7 Conclusão

Parabéns, agora você já pode brincar de escalar seu sistema tridimensionalmente e processar muitas vezes mais do que antes. Invenção melhor do que essa só duas dessa. Isso não é ótimo?

Figura 14.9: Sistemas reativos, não confundir com sistemas recreativos.

Não é bem assim que funciona. Ao escalar os serviços, aumentamos também sua disponibilidade e sua tolerância a falhas. O que é maravilhoso, mas, como afirma o fatídico teorema CAP, perderemos consistência.

Quando falamos de microsserviços, as siglas ACID e BASE dão vez à sigla CAP e o seu teorema nos lembra que a vida não é fácil e é repleta de escolhas.

Escolha aquilo que importa e sofra com isso. Exatamente o que eu fiz quando abri mão do meu futuro promissor como jogador de futebol. Decisão que o próprio futebol lamentará por décadas.

Como a maioria dos cenários baseados em microsserviços exigem disponibilidade e alta escalabilidade, conviver com a inconsistência é

algo natural. Não me refiro à ausência absoluta de consistência, mas da garantia de ter um nível satisfatório de confiabilidade. O nível de consistência depende do caso de uso que estamos implementando.

Uma dúvida que pode surgir é "como os aplicativos críticos contornam a ausência de consistência?". Como veremos nos próximos capítulos, existem técnicas que são usadas pela maioria das arquiteturas baseadas em microsserviços. Uma dúvida que permanecerá, entretanto, é sobre quem é melhor: eu ou o Messi? Essa, infelizmente, nem o tempo dirá.

Elasticidade e consistência são dois lados de uma mesma moeda. São assuntos profundos e de extrema importância. Em virtude disso, dedico os próximos capítulos à exploração desses conceitos.

CAPÍTULO 15
Inconsistência distribuída

Lidar com sistemas distribuídos é uma tarefa bastante difícil.

Pode haver vários componentes desenvolvidos por equipes diferentes durante longos períodos. A internet não é confiável. Erros humanos, como bugs de envio, parâmetros configurados incorretamente e falhas de hardware acontecem com desagradável frequência em grandes sistemas, não importa quão excelentes sejam as práticas de engenharia.

Além disso, mensagens imprevisíveis e permutações de todos os estados e condições possíveis tornam virtualmente impossível prever o que pode dar errado.

Falácias da computação distribuída

Muitos ficam sabendo que têm um sistema distribuído quando percebem que uma falha da qual nunca ouviram falar afeta todo o sistema. Até um sistema distribuído simples consegue não ser simples, pois, por definição, envolve mais de um nó, e os nós do sistema precisam se comunicar uns com os outros por uma rede.

Alerto que, até mesmo para criarmos pequenos sistemas distribuídos com apenas dois nós, fazemos certas suposições sobre como esse sistema funcionará.

Existe um grande problema sobre suposições: elas podem estar erradas. Algumas suposições sobre a computação distribuída são tão onipresentes e incorretas que receberam um nome. Estou falando, é claro, sobre as oito **falácias da computação distribuída**:

1. A rede é confiável.
2. Latência é zero.
3. A largura de banda é infinita.

4. A rede é segura.
5. A topologia não muda.
6. Existe um administrador.
7. Custo de transporte é zero.
8. A rede é homogênea.

Apesar de alguns argumentarem que, devido à evolução tecnológica e aos 30 e poucos anos decorridos desde sua concepção, as falácias se tornaram menos valiosas, elas continuam a nos alertar sobre a dificuldade da distribuição de sistemas.

Distribuir um sistema sem levá-las em consideração, o que não costuma ser raro, continua a ser um tiro no pé de qualquer cidadão. Não recomendo, mas, caso tenham coragem, não se sintam inibidos. É como costumo dizer: para quem tem coragem, toda praia é praia de nudismo. Boa sorte!

Para piorar, sistemas reativos exigem comunicação assíncrona entre os serviços. Como se não fossem suficientes as inconsistências decorrentes da computação distribuída e da distribuição de dados, a adoção de mensagens assíncronas traz consigo, como venda casada, uma leva de desgraças inconvenientes, como perdas de pacotes, reordenamentos, mensagens incompletas, chamadas recursivas potencialmente infinitas e retardos imprevisíveis. Podem soar inofensivas, mas essas desgraças são piores do que parecem e conseguem atrapalhar a vida de qualquer indivíduo, principalmente porque elas se camuflam e podem ser assustadoramente difíceis de detectar e prevenir.

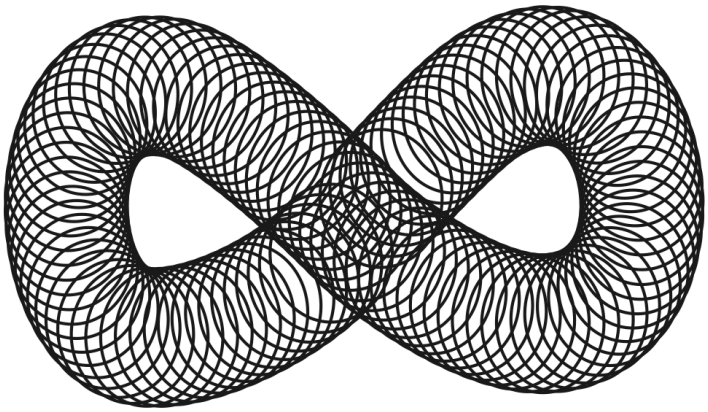

Figura 15.1: Sistemas reativos, não confundir com sistemas recursivos.

Felizmente, mas não tão felizmente quanto se não existissem, esses problemas são muito parecidos com uns velhos conhecidos de outro mundo da computação, a **programação concorrente**.

Para muitos efeitos, os sistemas distribuídos são sistemas concorrentes. Eles estão sendo executados em vários computadores executados em paralelo e compartilhando recursos.

15.1 Erros de fluxos concorrentes

Os problemas recorrentes na programação concorrente que podem ser encontrados, de maneira análoga, nos microsserviços são as **condições de corrida** com ou sem colisões, os badalados ***deadlocks*** e os ***livelocks***, que, embora sejam tão insolentes quanto o irmão da família *lock*, não detêm o mesmo status.

Assim como no contexto original, esses problemas podem acarretar graves consequências.

Diferentemente da programação concorrente, alguns deles não causam bloqueio de *threads* ou interrupção de processos. Nos

microsserviços, os serviços e suas *threads* continuam sendo executados normalmente. Mas, para um fluxo de dados específico, a operação pode ser bloqueada e/ou os valores envolvidos na transação podem ficar inconsistentes.

Condições de corrida

Assim como na programação concorrente, as condições de corrida são os tipos mais traiçoeiros. São difíceis de determinar, porque um fluxo de eventos pode ser executado corretamente centenas de vezes, por anos, sem nunca apresentar falhas. No entanto, se uma falha ocorrer, ela pode ser desastrosa.

Para piorar, quando se usa um depurador, o comportamento de agendamento dos programas tende a ser diferente, e os problemas de concorrência não dão as caras. Isso é chamado de *Heisenbug*, um bug de software que desaparece ou é diferente quando se tenta estudá-lo.

As condições de corrida podem originar as chamadas **colisões**. Elas ocorrem principalmente por erros de *timing*, quando em um infeliz e não planejado intervalo de tempo, duas ou mais operações acidentalmente sobrescrevem umas às outras.

Elas sinalizam quando o acesso a uma seção crítica deveria ser atômica.

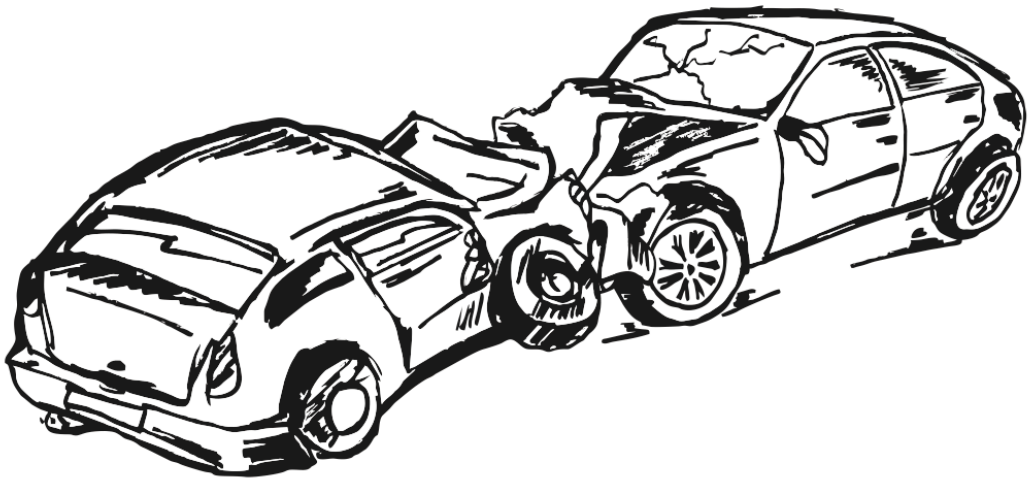

Figura 15.2: Erro de timing.

Digamos que, em um *e-commerce*, exista uma API que atualiza a quantidade de pedidos de um usuário. Considere que o valor esteja persistido em um banco de dados, portanto, para atualizar com o novo pedido, é necessário buscar o valor corrente, incrementar em 1 e persistir novamente.

Se mais de uma requisição for recebida em um lapso temporal muito estreito, ambas consultam e obtêm o mesmo valor de pedidos, por exemplo: 0, para um usuário que não tem pedidos ainda.

A primeira e a segunda atualizações atualizarão o valor para 1, porque as duas obtiveram o mesmo valor do banco de dados, enquanto o resultado esperado seria 2, o que seria uma inconsistência.

Outra forma de condição de corrida no mesmo cenário seria se existisse uma API específica para cancelamento de pedidos. Uma mensagem é emitida alertando a criação do pedido, o que incrementaria a contagem e, em seguida, outra é emitida informando que o pedido foi cancelado, o que decrementaria o total.

Se houvesse uma reordenação nas mensagens, muito comum, especialmente em comunicações assíncronas, e a mensagem de cancelamento de pedidos chegasse antes da mensagem de criação do pedido, não haveria pedido para ser cancelado.

Deadlocks* e *livelocks

Os *deadlocks* e *livelocks* são ocasionados quando há uma falha na elaboração da lógica no fluxo da comunicação entre serviços.

É um mal-entendido em que um participante espera um comportamento do outro e acaba se decepcionando, seja por excesso ou por omissão. Engraçado pensar que, na minha terra, a gente chama isso de casamento – talvez eu não sobreviva a essa piada.

Um conjunto de microsserviços está em *deadlock* quando cada microsserviço está esperando por um evento que só pode ser gerado por outro microsserviço do conjunto.

É o impasse gerado quando um serviço A espera um evento E_B de um serviço B para poder publicar o evento E_A, mas o serviço B só disparará E_B após receber E_A do serviço A.

Nenhum dos serviços envolvidos vai agir e o fluxo entrará em uma situação similar ao *deadlock*.

Em outras palavras, existe um impasse de expectativa e ninguém atua por esperar a iniciativa do outro. As mesmas condições de existência de *deadlock* em programação concorrente se aplicam a sistemas distribuídos.

Paul Krzyzanowski, em uma leitura sucinta (acessível em: https://www.cs.rutgers.edu/~pxk/417/notes/deadlock.html), conceituou e resumiu muito bem o que são os *deadlocks* distribuídos.

Para prevenir um *deadlock* distribuído, basta que uma dessas condições não seja satisfeita: 1) **Exclusão mútua:** todo recurso está ou associado a um único processo ou disponível; 2) **Posse e espera:** processos que retêm recursos podem solicitar novos recursos; 3) **Não preempção:** recursos concedidos previamente não podem ser forçosamente tomados; e 4) **Espera circular:** deve haver uma cadeia circular de dois ou mais processos, na qual cada um está à espera de recursos retidos pelo membro seguinte dessa cadeia.

Infelizmente, como em muitos outros aspectos dos sistemas distribuídos, os deadlocks são mais difíceis de detectar, evitar e prevenir.

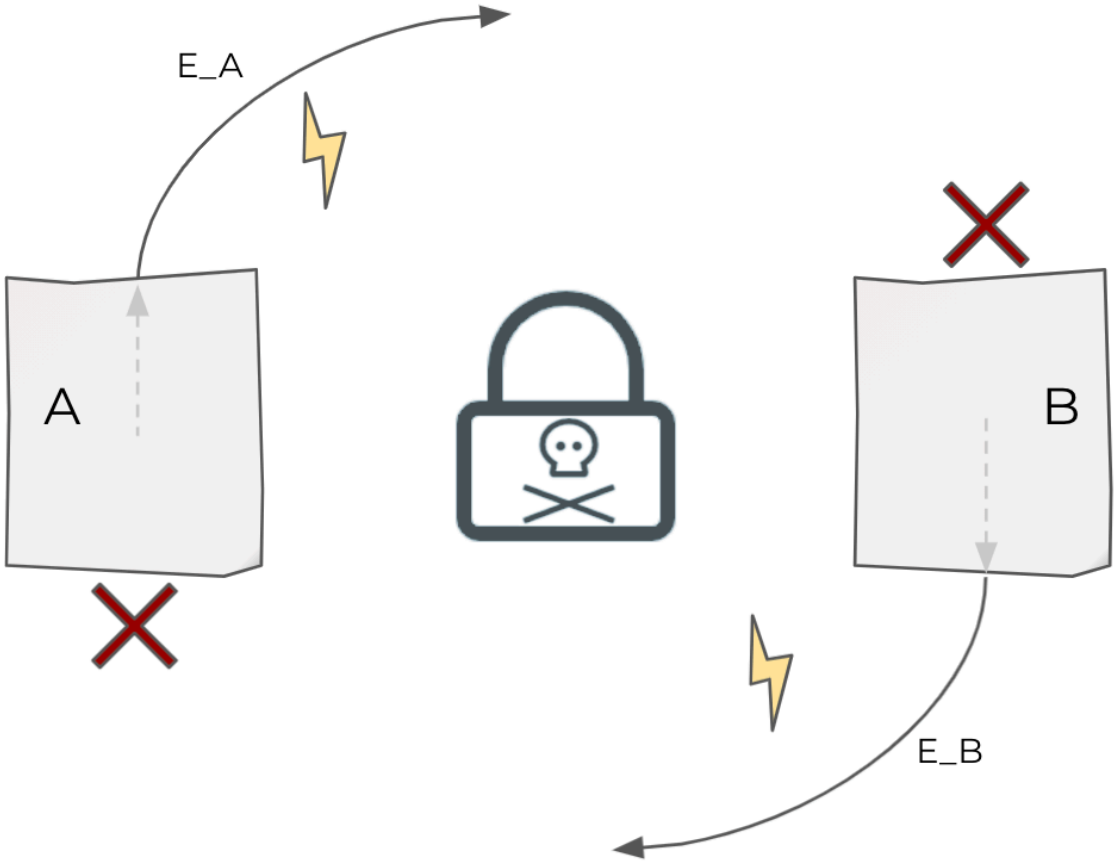

Figura 15.3: Deadlock.

Similarmente, quando um serviço `C`, ao consumir o evento `E_D`, emite `E_C` e, outro serviço `D`, ao escutar o evento `E_C` dispara `E_D`, estamos diante de um possível *livelock*, um *loop* infinito, que, infelizmente, assim como aparentam algumas piadas sem graça em certos livros sobre Sistemas Reativos, não acabam nunca.

Um bom exemplo de *livelock* é aquela história que comer dá sono e dormir dá fome. Nessa vibe, você fica eternamente preso nesse ciclo de comer e dormir. O que é delicioso, porém perigoso, por não possibilitar a alternação para outros estados.

Imagine que em um *e-commerce* um usuário deseja recuperar todas as informações relacionadas à sua conta. Para tal, por meio do seu aplicativo, uma requisição ao `serviço-de-contas` é criado. O `serviço-de-contas` requisita ao `serviço-de-dispositivos` informações sobre todos os dispositivos usados pela conta do usuário. O `serviço-de-dispositivos`, para completar as informações sobre todas as contas utilizadas pelos dispositivos, cria uma requisição ao `serviço-de-contas`, que, mais uma vez, precisa de dados do `serviço-de-dispositivos` e reinicia o ciclo interminável de requisições. O pobre coitado do usuário aguarda eternamente com o sabor amargo da desinformação.

Muitas vezes, identificar um laço infinito não é tão simples. Conforme os serviços crescem e se distribuem, mais fácil é criar chamadas recursivas.

Ter uma boa política de **timeouts** em todas as chamadas remotas e ser capaz de verificar **fluxos correlacionados** podem ajudar nessa situação treta. "Eu já estou na fila de chamadas dessa mensagem que estou processando?"

Diferentemente do *deadlock* distribuído, em um *livelock* os estados dos sistemas envolvidos estão constantemente mudando um em relação ao outro, mas, efetivamente, não há progresso.

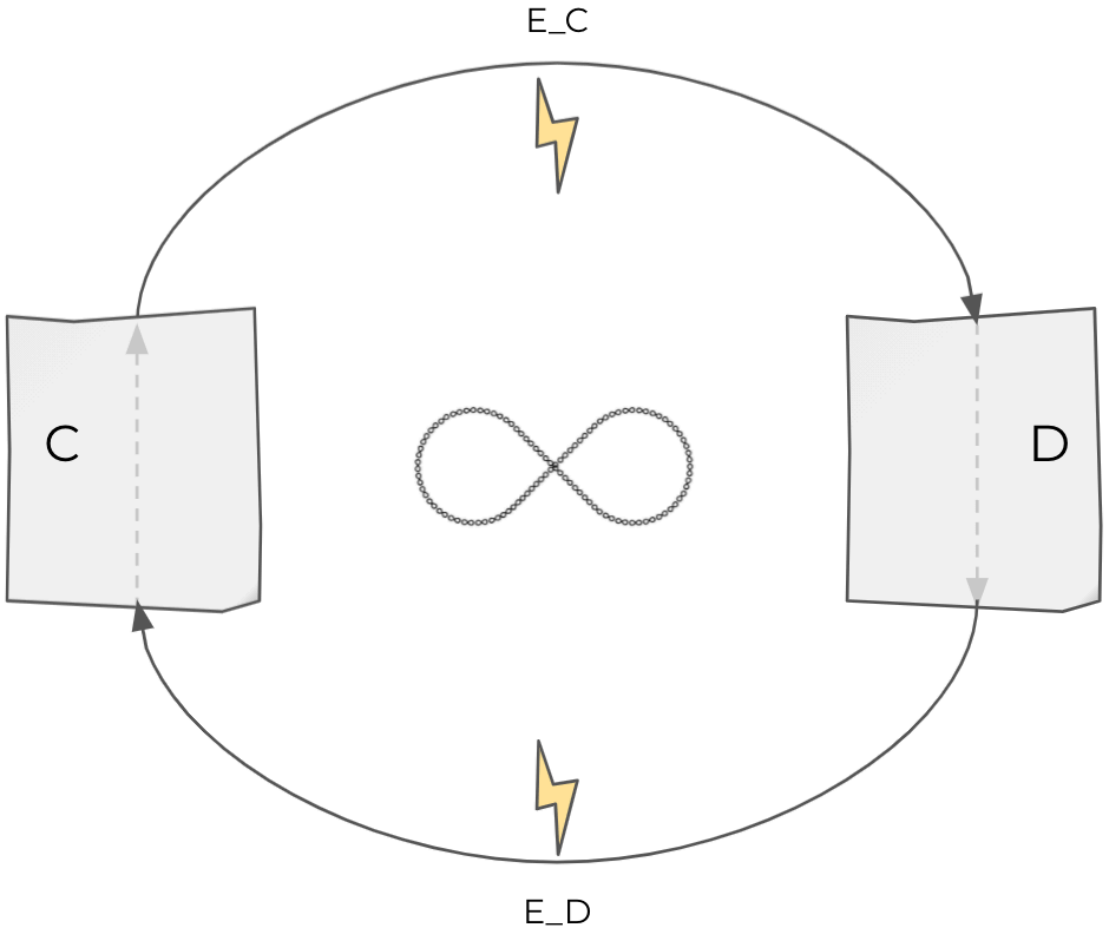

Figura 15.4: Livelock.

É válido salientar que tanto em *deadlocks* quanto em *livelocks*, havendo as outras condições necessárias, a quantidade de processos envolvidos é desimportante.

Como lidar com erros

Entender problemas concorrentes pode ser difícil. Solucionar pode ser ainda mais. Por isso, a abordagem mais fácil e mais amplamente usada para resolvê-los – carece de fontes – é não resolver e contar com a sorte de que esses problemas nunca deem as caras: a mais pura e santa ignorância. Mas essa pode ser uma alternativa terrível em alguns casos.

Caso ignorar as condições de corridas não seja uma opção válida, existem algumas abordagens baseadas em mecanismos de trava que priorizam a consistência em detrimento do desempenho. São elas a **trava otimista** e a **trava pessimista**.

A trava otimista utiliza uma chave de controle de versão, que é gerada e verificada após cada transação e só permite a atualização caso o sistema possua a última chave gerada. Se uma mudança aconteceu nesse meio tempo, você repete o processo até obter sucesso.

A trava pessimista exige que um terceiro processo qualquer bloqueie a modificação de dados enquanto a seção crítica de uma operação está em execução. É importante salientar que, se não usada com cautela, a trava pessimista pode não liberar o recurso protegido e ocasionar um *deadlock*.

Algumas soluções de SGBD, como é o caso do Amazon DynamoDB, fornecem recursos, como **expressões condicionais** para a manipulação de operações de escrita (criações, remoções e atualizações). Para essas operações, é possível especificar condições para determinar se os itens envolvidos devem realmente ser modificados. Se a condição for satisfeita, a operação é realizada, caso contrário, ela é interrompida. Expressões condicionais são exemplos de mecanismos de trava padrão desenvolvidos pela própria ferramenta.

Lamentavelmente, mecanismos de travas são infrutíferos com *deadlocks* e *livelocks* e, para piorar, podem ser sua causa raiz. Para esses problemas, a melhor forma de resolvê-los é sendo capazes de detectá-los e evitá-los antes que ocorram.

Depuração por log

Com essa finalidade, Jeff Langr, autor do livro *Agile Java* e contribuidor do livro *Clean Code* de Robert Martin, afirmou para mim que uma estratégia muito eficiente é a adição desenfreada de *logs*

para o aprimoramento da observabilidade. Afirmação que eu "assino embaixo". A prática também é conhecida como "depuração por *log*" ou "depuração estilo homem das cavernas".

Essa técnica é implementada adicionando instruções de *log*, sem medo de exagerar, em caminhos-chaves no código do aplicativo e imprimindo os *logs* em um arquivo local ou em um servidor remoto.

Quando o aplicativo é reimplantado sob a condição habitual e o problema investigado é reproduzido, as linhas de registro serão impressas. As impressões fornecem informações sobre o estado do aplicativo antes e logo após o problema. Simples, efetiva e à prova de *Heisenbugs*.

Ler os logs impressos ajuda o desenvolvedor a entender o que causou o problema e a evitá-los no futuro.

Alertas e gráficos podem ser criados e atrelados aos novos registros de forma que, ao serem acionados, os desenvolvedores tomem conhecimento rapidamente e possam entender qual situação está ocasionando o erro.

Nos próximos capítulos, veremos ferramentas e técnicas de observabilidade que detêm esse propósito.

A depuração por *log* permite que o aplicativo seja executado conforme o esperado, sem parar. É um ajuste natural para aplicativos nativos da nuvem e é considerado a única abordagem válida para depurar microsserviços e arquiteturas sem servidor.

No entanto, a depuração por *log* requer a adição de código para imprimir as instruções de *log* e desempenho, e o armazenamento e configuração adicionais são necessários para manter, armazenar e ler os arquivos de *log* impressos, o que pode gerar um aumento do custo do serviço.

O custo pode ser um fator impeditivo e pode não sanar todas essas dificuldades. Evitar completamente esses problemas é difícil e requer

a capacidade de prever com precisão os recursos e os tempos em que serão necessários, o que é praticamente utópico e não prático em sistemas reais.

Existem, por felicidade do destino, técnicas e princípios que minimizam sua ocorrência. Tais quais o **Princípio do Escritor Único** e o **ACID 2.0**.

15.2 Princípio do Escritor Único

Uma maneira simples de atenuar os problemas de condição de corrida é permitir que apenas um escritor seja capaz de alterar um certo dado. O raciocínio por trás da solução é simples: não há sobrescrita por concorrência se não houver concorrência.

Isolar o escritor e deixá-lo na solidão, à mercê da sua própria tarefa não é algo tão difícil. Falo com propriedade, escritores não são companhias tão prazerosas quanto se presume. Especialmente os metidos a engraçadinhos.

Figura 15.5: Escritor solitário.

Martin Thompson, um dos autores do Manifesto Reativo, cunhou o termo "Princípio do Escritor Único" (em https://mechanical-sympathy.blogspot.com/2011/09/single-writer-principle.html) em resposta ao uso em larga escala de bloqueios em ambientes simultâneos e às eficiências subsequentes que podemos obter com a consolidação de gravações.

O princípio evita esse problema, porque nunca precisa lidar com a gravação da versão mais recente de um item de dados que pode ter

sido gravado por outro.

Se você respeita esse princípio, cada escritor pode usar o tempo que quiser processando a lógica para seu propósito sem se preocupar com recursos para lidar com o problema de contenção.

> *O princípio do escritor único é que, para qualquer item de dados ou recurso, esse item de dados deve pertencer a um único contexto de execução para todas as mutações.* - Martin Thompson

Da perspectiva dos serviços, o princípio também se casa com a ideia de que os serviços devem ter uma única responsabilidade (*Single Responsibility Principle*) e um repositório exclusivo (*Single Repository Principle*).

Em sua essência, é um conceito simples. A responsabilidade pela propagação de eventos de um domínio específico é atribuída a um único serviço.

A fusão de gravações em um único serviço torna mais fácil gerenciar a consistência com eficiência. Esse princípio tem valor que vai além das propriedades de correção ou simultaneidade:

- Permite que verificações de consistência, como as travas otimistas e pessimistas, sejam aplicadas em um único lugar.
- Isola a lógica para a evolução de cada entidade comercial, no tempo, para um único serviço, facilitando a compreensão e futuras alterações. Por exemplo, implementando uma mudança de esquema de banco de dados.
- Não temos que duplicar esforços de análise ou desenvolvimento em outro microsserviço que pode emitir a mesma entidade.

O princípio incentiva equipes de serviço com propriedade definida de conjuntos de dados compartilhados e coloca o foco nos dados do lado de fora. Isso se torna importante em domínios que possuem

regras de negócios complexas associadas a diferentes tipos de dados.

Ben Stopford nos incendeia com o fogo da sapiência e, em seu magnífico livro *Designing Event-Driven Systems*, resume o princípio como:

> *A responsabilidade pela propagação de eventos de um tipo específico é atribuída a um único serviço.* - Ben Stopford

Outra estratégia que ameniza os problemas decorrentes da comunicação assíncrona, como reordenamentos, duplicações e retardos é fazer com que eles não afetem nosso sistema.

Uma das possibilidades é fazermos com que as mensagens sejam **idempotentes**, ou seja, eventuais duplicações seriam tratadas com indiferença. Se fizermos com que as mensagens sejam **comutativas**, um possível reordenamento de mensagens seria irrelevante.

Foi com esse conjunto de propriedades em mente que o termo **ACID 2.0** foi apresentado. Sim, você não leu errado. Senhoras e senhores, é com uma enorme alegria que vos apresento o *ACID 2.0: o retorno*.

15.3 Lá e de volta outra vez: ACID 2.0

Figura 15.6: Matemática ácida.

Para explicar o conceito do ACID 2.0, recorro a outra matéria da grade curricular do ensino brasileiro, um dos mais conhecidos ramos da matemática: a adorável álgebra.

O que a matemática do ensino básico tem a ver com computação em nuvem e sistemas distribuídos? Permitam-me aprofundar.

Uma operação algébrica possui propriedades. Considere a multiplicação, por exemplo. Podemos afirmar que é uma operação **associativa**, **comutativa** e **distributiva**. Dependendo dos fatores envolvidos na operação, pode-se dizer também que é **idempotente**:

- **Associativa:** a precedência das operações que envolvem apenas multiplicações é irrelevante. Assim, $A \times (B \times C) = (A \times B) \times C$. Por exemplo: $2 \times (3 \times 4) = (2 \times 3) \times 4 = 24$.
- **Comutativa:** a ordem dos fatores não altera o produto. Logo $A \times B = B \times A$, por exemplo: $2 \times 3 = 3 \times 2 = 6$.
- **Idempotência:** pode ser aplicada vária vezes sem que o valor do resultado se altere após a primeira aplicação. Assim, $A \times B = A \times B \times B = A \times B \times B \times B$. A multiplicação é idempotente apenas quando um dos fatores é 0 ou 1. Por exemplo: $10 \times 1 = 10 \times 1 \times 1 = 10 \times 1 \times 1 \times 1 = 10$, assim como $10 \times 0 = 10 \times 0 \times 0 = 10 \times 0 \times 0 \times 0 = 0$.
- **Distributiva:** uma soma multiplicada por um número é igual à soma dos produtos entre esse número e as parcelas da soma. Logo $(A + B) \times C = A \times C + B \times C$, por exemplo: $(2 + 3) \times 4 = (2 \times 4) + (3 \times 4) = 20$.

Interessante, não? Pat Helland e Dave Campbell (em: https://database.cs.wisc.edu/cidr/cidr2009/Paper_133.pdf) acharam o mesmo e resolveram aplicar essas mesmas propriedades para o design de protocolos eventualmente consistentes.

Assim nasceu o ACID 2.0, que se trata de um conjunto de regras para o design de soluções eventualmente consistente e dá uma pequena cutucada no ACID tradicional. Enquanto as propriedades originais do ACID tratam de correção e precisão, as novas são de flexibilidade e robustez.

Os autores do estudo certificam que ACID 2.0 é sobre fornecer instruções para que infortúnios previamente citados, como reordenações, duplicações e atrasos não ofereçam riscos de consistência ao sistema.

Segundo eles, um sistema é compatível com as normas ACID 2.0 quando 1) as etapas individuais acontecem em um sistema ou mais; 2) o aplicativo é explicitamente tolerante com o trabalho fora de ordem; e 3) os sistemas também permitem tarefas sendo executadas mais de uma vez por máquina.

As propriedades do ACID 2.0 fornecem aos sistemas a flexibilidade necessária para distribuir, paralelizar operações e torná-los assíncronos e, potencialmente, reativos.

Assim como na álgebra elementar, as propriedades são associatividade, comutatividade, idempotência e distributividade que, em conjunto, pela união dos seus poderes, invocam o ACID 2.0:

Figura 15.7: O poder é de vocês!

- **Associativo** `2 × (3 × 4) = (2 × 3) × 4` **:** capacidade de agregar conjuntos de dados. A existência do agrupamento de

mensagens não impacta o resultado final, portanto permite o **processamento em lotes**.
- **Comutativo** `5 × 6 = 6 × 5` **:** a ordem das mensagens é irrelevante.
- **Idempotente** `10 × 1 = 10 × 1 × 1` **:** suporta a duplicação de mensagens.
- **Distribuído** `(2 + 3) × 4 = (2 × 4) + (3 × 4)` **:** indica se uma operação é tolerante à ordem parcial. Existe uma forte teoria nos corredores escusos da computação que essa propriedade foi incluída apenas para fazer o acrônimo ACID funcionar. Não me pronunciarei a respeito.

Suponha que eu precise processar um grande conjunto de registros para calcular a soma de certos elementos em todos os registros. A associatividade permite que essa operação seja agrupada em vários pedaços e com qualquer combinação entre eles. Assim, poderíamos processar cada pedaço em uma máquina separada e, em seguida, agregar os resultados intermediários na soma final.

A idempotência se traduz na operação que pode ser chamada repetidamente sem alterar o resultado. Algumas operações, como ler um valor ou definir um campo, são inerentemente idempotentes. Outras podem se tornar idempotentes usando conceitos, como **Identificadores de Correlação**, tópico já mencionado no capítulo *Mensagens distribuídas*. A operação só é efetuada se tivermos certeza de que o identificador da transação é inédito.

A comutatividade resolve o problema da reordenação. Por exemplo, em alguns contextos, uma operação de exclusão deve abordar um objeto obrigatoriamente introduzido por uma operação de inserção anterior. Essas operações estão causalmente relacionadas. A comutatividade permitiria a aplicação da exclusão antes da inserção.

A propriedade da distribuição é tautológica e detém um papel-chave nos sistemas distribuídos. Podemos argumentar que, sem a distribuição, os sistemas distribuídos seriam conhecidos apenas

como sistemas. De certa forma, isso me remete àquela "piada" que diz que a comida japonesa, no Japão, é chamada apenas de comida.

A classificação ACID 2.0 nos fornece meios práticos de rotular operações em diferentes níveis de abstração (mensagens, fluxos e sistemas) em ambientes distribuídos e não tem nada em comum com o ACID clássico que conhecemos do SGBD.

A primeira versão é reservada aos SGBDs e suas transações, enquanto a segunda é reservada aos sistemas distribuídos e nos orienta a garantir algum nível de consistência sem quaisquer mecanismos de coordenação adicionais.

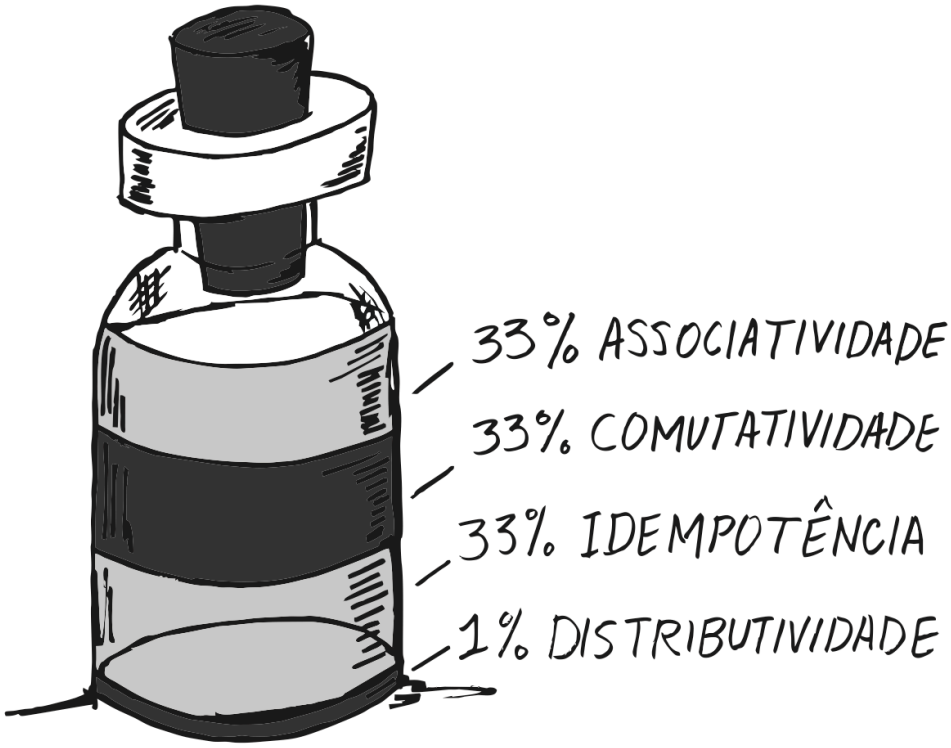

Figura 15.8: ACID 2.0: agora é de com força.

15.4 Conclusão

Neste capítulo, reforço a ideia de que distribuir um sistema é a centelha necessária para que problemas típicos da programação concorrente deem as caras e incendeiem também os sistemas distribuídos.

Alguns problemas, como atrasos e reordenações de mensagens, ocorrem e podem causar grandes confusões. Podem parecer casos improváveis à primeira vista, pelo menos tão raros quanto encontrar alguém que tenha comprado a licença do Winrar.

Mas multiplique a improbabilidade pela magnitude de mensagens que um *e-commerce* processa e você verá que a quantidade de inconsistências não será desprezível.

Soluções simples, como manter a decisão de assegurar que um dado seja alterado apenas por um serviço, podem ser suficientes para assegurar um maior nível de consistência.

Vimos também que existem instruções que nos orientarão a atenuar os efeitos advindos da distribuição dos sistemas. Esse conjunto de instruções, seja por falta de criatividade, seja por provocação, ou pelos dois, é chamado de ACID 2.0 e será referenciado no infernal próximo capítulo.

Capítulo 16
O inferno da consistência

Não se preocupem, projetar sistemas distribuídos consistentes é como andar de bicicleta no parque. Só que o parque está pegando fogo, a bicicleta está pegando fogo e sua companhia é o diabo, porque sistemas distribuídos consistentes são o inferno!

Figura 16.1: Projetar sistemas distribuídos consistentes.

Os microsserviços apresentam problemas de consistência devido à louvável insistência no gerenciamento descentralizado de dados, à alta disponibilidade e à tolerância a falhas. Como consequência, desenvolvedores devem prevenir ou detectar quando as coisas estão sem sentido antes que seja tarde demais.

16.1 O que é consistência

O termo "consistência" é amplamente utilizado dentro e fora da indústria do desenvolvimento de software. Existem diferentes significados a depender do contexto e da conveniência de quem usa.

A irônica falta de consistência na terminologia representa um reflexo da profundidade do assunto e vai muito além do escopo deste livro. Ainda assim, uma breve explicação se faz necessária para que possamos nos aprofundar em outros assuntos.

Apesar da confusão intrínseca em torno do tema, todos temos uma noção intuitiva aproximada do que é e o que facilita bastante minha vida enquanto transmissor de conteúdo. Essa intuição sobre o tópico baseada nas experiências de vida geralmente envolve conceitos similares aos de transações de dados. O que é ainda melhor.

Pyong Lee, ex-BBB, subcelebridade instantânea e youtuber especializado em hipnose e ilusionismo – que acredito não possuir propriedade nenhuma para falar de conceitos computacionais – declara, em um dos seus vídeos, que a consistência é bastante atrelada à ideia de se manter sempre atualizado:

> *Não é porque você já tem domínio sobre um assunto que não pode continuar se atualizando. A consistência está, portanto, na atualização contínua. Tenha sempre em mente que parar é um convite à estagnação e que sempre haverá alguém que não parou de se atualizar.* - Pyong Lee.

Boa parte da declaração citada é irrelevante para o livro, mas, de maneira geral, corrobora com meu ponto de que todos temos uma intuição bastante aproximada do que é consistência, que, na computação, costuma ser definida como:

Consistência é uniformidade ou conformidade na aplicação de algo, por uma questão de lógica ou necessidade.

É interessante pensar que alguém tão longe da computação constata algo que, ao reformularmos, removermos o secundário, e adornarmos o restante com o véu do desenvolvimento de software, pode ser interpretado como: consistência está na continuidade, está em continuar se atualizando.

A pergunta "por que um hipnotizador não hipnotizou todos os participantes do BBB e ganhou o *reality show* sem dificuldades?", no entanto, se juntará a outras grandes perguntas não respondidas pela humanidade, como, por exemplo, o que os criadores de *Game of Thrones* tinham na cabeça quando escreveram a última temporada?

16.2 Consistência no mundo distribuído

Figura 16.2: Você tem um segundo para ouvir a palavra da consistência?

Ao construir arquiteturas tradicionais de microsserviços, os serviços que encapsulam os dados do domínio gerenciam as mudanças concorrentes, isoladas dos outros sistemas.

Grupos de serviço não costumam ter esse controle, a não ser que todos os serviços se coordenem ao redor de apenas um centro de dados e um modelo de consistência global, o que contrariaria o princípio de Repositório Único, um preço alto para os princípios pregados pelos sistemas reativos.

Os conjuntos de serviços podem chamar uns aos outros de modo síncrono, mas, conforme os dados transitam entre serviços,

frequentemente se tornarão defasados dada a imprevisibilidade da cadeia de comunicação.

Uma artimanha diabólica comum é criar um sistema distribuído simplesmente traçando um paralelo com aplicações não distribuídas. Essa abordagem, inicialmente, fornece um modelo muito intuitivo enquanto tudo progride com uma curva aparentemente linear, mas, quando os serviços começam a se reproduzir feito demônios no cio e a curva de complexidade mostra sua verdadeira face, percebe-se que operar os sistemas que seguem essa fórmula se torna uma tarefa virtualmente inconcebível. Reza a lenda que há um lugarzinho reservado no céu para pessoas que tentaram e falharam miseravelmente nessa abordagem. Se é verdade ou não, não sei. Meu compromisso é com a notícia.

Figura 16.3: Sistemas reativos, não confundir com sistemas reprodutivos.

Recordem que os vacilos que os desenvolvedores costumam cometer ao migrar para um cenário distribuído são tão gritantes e frequentes que culminaram na criação das *Falácias da Computação Distribuída*, apresentadas no capítulo passado.

Os sistemas reativos conseguem a proeza de dificultar ainda mais essa missão. Eles eliminam deliberadamente a necessidade de estado global e evitam execuções síncronas. Como consequência, esses sistemas são frequentemente referidos como **eventualmente consistentes**.

16.3 Modelos de consistência

Consistência é mesmo um assunto cabuloso nos sistemas distribuídos e engloba uma abundância de modelos, variando em complexidade, força, escopo e ponto de vista. Se mal gerenciada, é capaz de fazer o próprio Tinhoso abandonar o inferno para não conviver com os problemas que podem resultar.

Como exemplos dessa abundância de modelos, listo a consistência estrita, consistência sequencial, consistência contínua, inconsistência aceitável, consistência lenta, consistência geral, consistência causal, consistência local etc. Já deu para capturar o espírito tenebroso da coisa, né?

Por conta dessa vasta lista e por uma tentativa, geralmente frustrada da minha parte, de não fugir do assunto principal, me limitarei a discutir sobre três tipos de consistência.

Consistência forte

Após a execução, qualquer acesso subsequente fornecerá o valor atualizado. É dessa consistência, no contexto de banco de dados, que o acrônimo ACID fala. Não há margem para valores semicorretos. A consistência forte oferece uma garantia de **linearização**.

Linearizabilidade refere-se a atender solicitações simultaneamente. As leituras têm a garantia de retornar a versão confirmada mais recente de um item. Um cliente nunca vê uma gravação não

confirmada ou parcial. Os usuários sempre têm a garantia de ler a última gravação confirmada.

Muito linda, né? Pena que é inalcançável nos microsserviços e, às vezes, até nos monólitos ácidos também.

Mesmo com um único SGBD ácido local pode haver inconsistências, dependendo de como os acessos concorrem entre si. Por mais que, no banco de dados, o valor permaneça consistente, a forma como é utilizado pode apresentar informações incoerentes. O que não é necessariamente um problema, visto que, contraditoriamente, a informação não precisa permanecer consistente quando atinge um usuário.

A consistência forte requer coordenação, o que é muito caro e coloca um limite superior na escalabilidade, disponibilidade e rendimento. A necessidade de coordenação significa que os serviços não podem progredir individualmente enquanto esperam o consenso.

Figura 16.4: Consistência forte.

Consistência fraca

O sistema não garante que os acessos subsequentes obterão um valor atualizado. Enquanto um conjunto de condições não for satisfeito, todos os acessos devolverão um dado desatualizado.

O período entre a atualização e o momento em que o valor é, efetivamente, atualizado, ou seja, quando as condições são satisfeitas, é designado de **janela de inconsistência**.

Busca-se minimizar esse período e, se não ocorrer nenhum erro, o tamanho máximo da janela de inconsistência pode ser estimado,

com base em fatores, como atrasos de comunicação, carga no sistema e número de sistemas envolvidos na transação.

Uma forma específica de consistência fraca é a **consistência eventual**. Essa forma diz que, se nenhuma nova atualização for feita, eventualmente todos os acessos retornarão o valor atualizado.

As transações BASE e a consistência do teorema CAP, em sua maioria, são baseadas nessa força de consistência.

Figura 16.5: Consistência fraca.

Consistência causal

Consistência causal é um modelo de consistência eventual que possui menos garantias, mas poderia ser mais eficiente e simples de implementar. Nesse caso, temos uma garantia bastante simples: apenas as **operações causais**, as operações que possuem relação causa-consequência, aparecem em ordem e eles aparecem em ordem para todos os clientes.

No entanto, outras operações que não possuem vínculos causais podem ser vistas por clientes diferentes em ordem arbitrária.

Por comparação, a consistência forte exige que todas as operações confirmadas apareçam na mesma ordem. Então, estamos relaxando essa restrição para se aplicar apenas a operações relacionadas.

Se um serviço realizar uma operação de gravação A e outro, ao observar A, realiza uma operação B, então é possível que A seja a causa de B. Nesse caso, podemos dizer que A **potencialmente causa** B. A consistência causal garante que, se A potencialmente causa B, então B foi potencialmente causado por A.

Considere uma conversa entre amigos sobre o jogo Fluminense x Flamengo em um aplicativo de mensagens:

1. **Torcedor do Flamengo:** oh não, o Fluminense fez um gol e está ganhando do Flamengo.
2. **Torcedor do Flamengo [alguns minutos depois]:** que beleza, o Flamengo fez vários gols e passou a ganhar no jogo.
3. **Torcedor do Fluminense:** que lástima, tudo estava indo tão bem.

Há uma relação causal entre a mensagem **2** e **3**. O torcedor do Fluminense lamenta (3) porque o adversário marcou gols (2). A violação de causalidade poderia originar inconsistências, como um torcedor do Fluminense lamentar um gol do próprio time.

Veja a seguir que a mensagem **1**, seguida da mensagem **2**, dá a entender que o torcedor do Fluminense não curtiu que o próprio time tenha feito um gol.

1. **Torcedor do Flamengo:** oh não, o Fluminense fez um gol e está ganhando do Flamengo.
2. **Torcedor do Fluminense:** que lástima, tudo estava indo tão bem.
3. **Torcedor do Flamengo [alguns minutos depois]:** que beleza, o Flamengo fez vários gols e passou a ganhar no jogo.

A consistência causal garante que esse tipo de cenário não ocorra. Não existe forma mais forte de consistência que também garanta baixa latência.

A consistência causal captura as relações causais potenciais entre as operações e garante que todos os processos observem as operações causalmente relacionadas em uma ordem comum.

Em outras palavras, todos os processos no sistema concordam com a ordem das operações causalmente relacionadas e podem discordar na ordem das operações que não são causalmente ligadas.

Figura 16.6: Não é casual, é causal.

Claro, isso é apenas um arranhão na superfície, existem muitos outros modelos de consistência para serem discutidos, mas esses são suficientes por agora.

16.4 Conclusão

Durante a deliciosa jornada do capítulo, partindo do paraíso em direção ao mundo inferior, perdemos gradualmente a esperança de ter um sistema reativo e consistente. Tornou-se quase impossível ser feliz.

Não é fácil se conformar, mas a consistência forte não faz parte do pacote de assinatura dos microsserviços.

Um servidor e cliente, por exemplo, só podem ser consistentes enquanto estão conectados um ao outro. Caso contrário, por não haver possibilidade de atualização, segundo o ex-BBB Pyong Lee, estarão inconsistentes e retornarão, eventualmente, ao estado de consistência após uma nova conexão.

Ainda assim, ser capaz de funcionar *off-line* é uma funcionalidade desejável, assim como sincronizar novamente quando a conexão é restabelecida. A utilidade desse modo de operação depende de um trabalho específico da finalidade.

Um aplicativo de compras pode permitir que você selecione itens para serem comprados, mas não pode prosseguir caso os itens não estejam disponíveis.

Contudo, vimos também que, se nos conformarmos que a consistência é superestimada, podemos trabalhar com o que se tem em mão e alcançar objetivos preciosos.

Figura 16.7: Morre, diabo.

Isso nos leva à necessidade de adotar a inconsistência como filha, chamá-la de bebê e pensar em como projetar sistemas confiáveis apesar de sua inevitável presença.

Desenvolver sistemas reativos é isso. É abrir mão da consistência a todo custo e perceber que, no fim das contas, ser consistentemente ruim é muito melhor do que inconsistentemente bom.

Cansei de ser iludido

Os microsserviços vêm com uma série de vantagens e desvantagens. A consistência enfraquecida é uma desvantagem com a qual você terá que conviver. É tipo o Jar Jar Binks em Star Wars, uma oferenda indesejada que você se conforma por conta do todo.

Figura 16.8: Darth Binks.

Quando falamos de sistemas reativos, a inconsistência é um inferno na terra, pois não se trata de "se acontecer", mas sim de "como projetar aplicações que a suportam".

Precisamos confiar na **consistência eventual**. Graças a ela, a resiliência, a elasticidade, o isolamento e outros pontos positivos, são possíveis. Diferentemente do Jar Jar Binks, que é só ponto negativo. Ladeira abaixo do início ao fim.

A consistência eventual nos permite elevar o máximo do que pode ser feito em termos de escalabilidade, disponibilidade e acoplamento reduzido. Além disso, como disse um amigo meu de forma despretensiosa em uma reunião de trabalho: a consistência eventual é boa, porque soa bem e não parece que algo está inconsistente.

Às vezes, confiar na inconsistência não é permitido, porque pode nos forçar a desistir muito da semântica de negócios de alto nível.

Portanto, se for esse o caso, usar consistência causal pode ser uma boa opção, aliás, é o melhor que podemos fazer em um sistema sempre disponível e escalável.

Parte 6: Resiliência

Capítulo 17
Fluxos resilientes

Figura 17.1: DJ Virgs.

Este capítulo se tornou bem maior do que originalmente eu previa – acredito que o Renato Russo falou o mesmo após a concepção de *Faroeste Caboclo*.

Para compensar e tornar a leitura menos enfadonha, pretendo promover uma experiência sinestésica. Embalarei as próximas páginas com ritmos envolventes e ecléticos do *pop music*, todos excelentes para dar uns beijinhos, diga-se de passagem.

Para dar início à experiência e começar a falar sobre disponibilidade no próximo parágrafo, eu, sob a alcunha de DJ Virgs, solicito ajuda da eterna Alcione e seus versos desesperados de *Você me vira a cabeça*:

> *Só pra ter alguém*
>
> *Que vive sempre ao seu dispor*
>
> *Por um segundo de amor*
>
> – Alcione - *Você me vira a cabeça.*

A letra nos dá uma verdadeira aula sobre dedicação e disponibilidade. Chega a arrepiar.

17.1 Disponibilidade

Quando um serviço invoca outro, não se pode desconsiderar as chances de o outro não estar disponível ou demorar muito para responder, o que levaria ao esgotamento dos recursos do serviço solicitante ou ao entupimento do que recebe as chamadas, tornando ambos, eventualmente, indisponíveis.

A disponibilidade é um importante atributo de qualidade de software. Ela expressa a quantidade de tempo pelo qual um componente está à disposição, em comparação com a quantidade de tempo pelo qual o componente deveria estar disponível.

A disponibilidade pode ser expressa pela seguinte fórmula:

disponibilidade = tempo ativo ÷ (tempo ativo + tempo ocioso)

Percebemos que há dois modos de aumentar a disponibilidade: 1) aumentar o **tempo ativo**, e 2) reduzir o **tempo ocioso**.

Ao passo que abordagens tradicionais visam aumentar o tempo de atividade, aplicações resilientes primam por reduzir o tempo ocioso. Mecanismos, como fraco acoplamento, o isolamento, o controle de latência e a supervisão não são suficientes para a resiliência reativa.

É fundamental construir componentes robustos que possam tolerar erros dentro de seu escopo, tratar falhas de outros componentes dos quais eles dependem, lidar com as falhas totais ou imparciais em vez de tentar evitá-las a todo custo e ficar indisponível por muito tempo quando elas ocorrerem.

17.2 Defeitos, erros e falhas

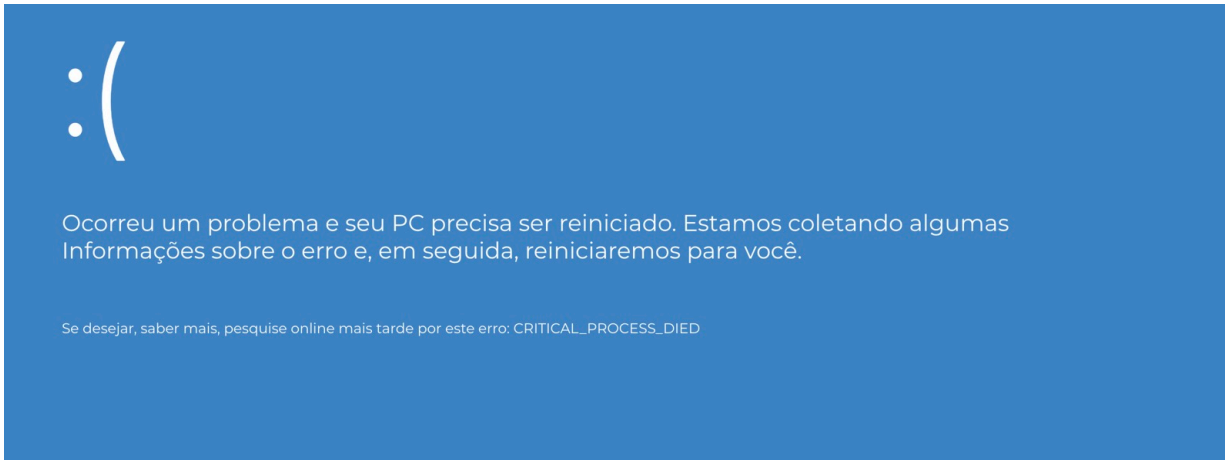

Figura 17.2: Exemplo relativamente conhecido de falha.

Como todo objeto de estudo, existe uma linguagem especializada associada à tolerância a erros.

Você vai se surpreender com a quantidade de sinônimos que a palavra "erro" tem: defeito, falha, incidente, problema, equívoco, engano, incorreção, aberração, desacerto, irregularidade, NX Zero, calças saruel, tatuar "carpe diem", gente que usa pochete, comer ovo com gema mole e muitos outros.

Em conjunto, eles vão, aos poucos, minando completamente o bem-estar de uma sociedade já tão debilitada quanto a brasileira. A única explicação plausível para essa quantidade de sinônimos é a de que nós, enquanto humanos, erramos muito e variamos a palavra para não cair na mesmice.

Essas belas palavras que eu mesmo escolhi me ajudam a introduzir a segunda música dessa depressiva *playlist*:

> *Give me reason, but don't give me choice*
>
> *Because I'll just make the same mistake again*
>
> – James Blunt - *Same mistake.*

Apesar de, geralmente, resumi-los em bug, os termos **defeito**, **erro** e **falha** merecem atenção especial por possuírem conotações importantes no ramo do desenvolvimento de software.

Robert S. Hanmer, no livro *Patterns for Fault Tolerant Software*, assevera:

Erros, defeitos e falhas

Uma **falha** de sistema ocorre quando o resultado do serviço não está em conformidade com a descrição acordada.
Um **erro** é a parte do sistema que pode levar a uma falha subsequente; um erro que afeta o serviço é uma indicação de que uma falha ocorre ou ocorreu.
A causa julgada de um erro é um **defeito**.

– Robert S. Hanmer.

"Por que dar rótulos diferentes para um bug?", você pode me perguntar. Identificar corretamente os termos simplifica o tratamento da falha. Eles ajudam a comunicar qual a precisão do seu conhecimento do problema.

Falhas transientes

As falhas transientes ocorrem durante a comunicação com um componente ou serviço externo, por motivos, como falha na rede ou sobrecarga do servidor.

No nosso cotidiano, na hospedagem em nuvem, por exemplo, onde os aplicativos usam serviços que competem por recursos e largura de banda e se comunicam pela internet, todo tipo de erro acontece e as chances de obtê-los são grandes.

Falhas como perda momentânea de rede, sobrecarga de serviço, interrupções de banco de dados, *timeouts* mal configurados e indisponibilidade de serviço se tornaram, comparativamente, mais proeminentes.

Esses problemas são efêmeros e podem ser facilmente contornados simplesmente tentando novamente após um certo tempo, por isso essas falhas também são chamadas de falhas transitórias. Mais adiante, veremos que chamamos essa técnica de *retry*.

É o momento perfeito para dizer, como cantam os lábios do eterno Maluco Beleza e que agora são os lábios do livro de Sistemas Reativos, "tente outra vez".

> *Não diga que a canção está perdida*
>
> *Tenha fé em Deus, tenha fé na vida*
>
> *Tente outra vez!*
>
> – Raul Seixas - *Tente outra vez.*

Figura 17.3: Toca Raul.

Os sistemas reativos aceitam as falhas como elas são. Elas acontecem mais do que gostaríamos e não temos tanto controle

quanto gostaríamos sobre elas.

Ainda assim, as falhas devem ser reprimidas, traduzidas como mensagens e enviadas para componentes supervisores com a finalidade de monitorá-las em contexto seguro. Para aprimorar a resiliência, os sistemas reativos devem resistir aos erros e fazer com que eles sejam parte do sistema, sem preconceito e discriminação.

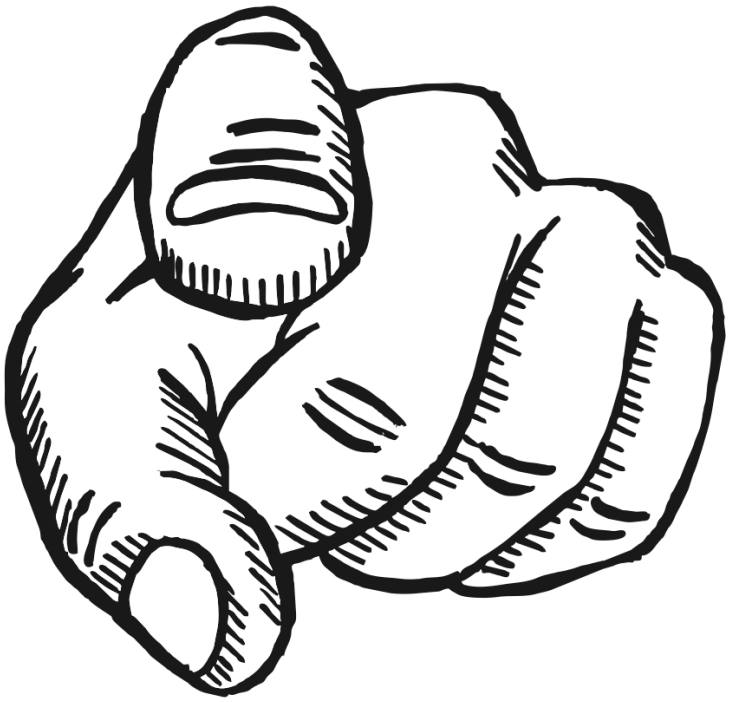

Figura 17.4: Sistemas reativos, não confundir com sistemas recriminativos (sic).

17.3 Padrões de resiliência

Resiliência é a capacidade de um sistema de lidar e se recuperar de falhas tanto inadvertidas quanto maliciosas. É a habilidade do sistema de tratar situações inesperadas sem que o usuário sequer perceba, no melhor caso. Logo, a detecção de falhas e a recuperação rápida e eficiente são necessárias para manter a resiliência.

A resiliência desempenha um papel-chave nos sistemas reativos, figurando como uma das quatro propriedades mencionadas no Manifesto Reativo.

Isso significa que sistemas reativos devem compreender as falhas a todo custo. Devem tratá-las como prioridade e desenvolver a arquitetura assumindo sempre que elas ocorrerão. Devem desenvolver uma arquitetura Orientada a Falhas, isto é, ao decorrer do desenvolvimento, o fluxo da falha deve ter prioridade tão grande quanto o próprio fluxo do sucesso.

Embora você possa investir um tempo considerável escrevendo sua própria estrutura de resiliência, esses produtos já existem e são chamados de **padrões de resiliência**.

Existem vários padrões de resiliência disponíveis pela internet afora, provando que o assunto é pop – assim como o Papa, a desgraça alheia e as pessoas que vestem camisas de futebol fora de um estádio.

Figura 17.5: O pop não poupa ninguém.

Os padrões de resiliência são a primeira linha de defesa do aplicativo. Eles têm como foco os mecanismos e estruturas que podem ser adicionadas na solução para habilitar sua operação em casos de falhas internas.

Eles são relativamente simples de aprender e aplicar e ajudam a fazer com que um usuário não desconfie de que uma falha inesperada ocorreu, ou pelo menos que ele consiga continuar a usar a aplicação com um escopo reduzido mas funcional. O que é conhecido como a **degradação graciosa**.

Sobre os padrões de resiliência, Uwe Friedrichsen, no seu artigo *Towards Successful Resilient Software Design* (https://www.infoq.com/articles/towards-resilient-software-design/), relata:

"Os padrões de resiliência aumentam os custos de implementação, a complexidade da solução. Isso é crítico, pois a complexidade é inimiga da robustez. Quanto mais complexa se torna a solução, menos compreensível ela se torna e mais provável que surjam efeitos inesperados que afetarão a robustez.

Portanto, é importante não implementar tantos padrões quanto possível, mas encontrar o ponto ideal de robustez entre medidas de resiliência insuficientes e complexidade excessiva."

– Uwe Friedrichsen

Ou seja, por mais bem-intencionados que sejam os padrões de resiliência, usá-los inadvertidamente pode causar o efeito contrário: diminuir a resiliência e, no extremo, matar sua solução de software.

Figura 17.6: Nossa, nossa, assim você me mata.

Existem diversos padrões, uns mais famosos do que outros e, como as páginas deste livro e a paciência do leitor/leitora são bens finitos, descreverei os que considero mais populares e possuem uma interdependência lógica.

Padrão	Resumo
Tempo limite (*Timeout*)	Limita a duração pela qual um serviço pode aguardar uma mensagem
Solução alternativa (*Fallback*)	Define o comportamento substituto após uma falha
Tentar novamente (*Retry*)	Configura tarefas de repetição em operações designadas
Disjuntor (*Circuit breaker*)	Bloqueia as operações solicitadas por um período quando as falhas estão em alta

Para exemplificar a funcionalidade dos padrões, utilizaremos um exemplo de *e-commerce* e transações causalmente relacionadas.

Imagine um `serviço-de-pagamento` como parte de uma plataforma de compras. Quando um cliente deseja realizar um pagamento, o `serviço-de-pagamento` deve verificar se não há nenhuma intenção fraudulenta. Para fazer isso, ele conta com o auxílio do `serviço-antifraude`.

Para verificar a transação, o `serviço-de-pagamento` envia um comando com os dados do pagamento para o `serviço-antifraude`. Se tudo ocorrer como esperado, receberá um evento como resposta, indicando se a transação é fraudulenta ou não.

O que acontece se o `serviço-antifraude` não estiver respondendo ou demorar muito para responder?

Tempo limite (*Timeout*)

O padrão *timeout* é bastante simples. O objetivo é evitar erros por tempo de espera ilimitado. Logo, quando a mensagem aguardada não chega no limite preestabelecido, o fluxo deve ser tratado como falha e gerenciado a partir de então.

É não aceitar desculpas quando o comportamento esperado não ocorreu a tempo, como o impiedoso portão do Enem. Ou, como diria *One Republic* em *Apologize*:

> *It's too late to apologize.*
>
> *It's too late*
>
> – One Republic - *Apologize.*

Em mensagens do tipo consulta com protocolos síncronos, como o http, o próprio protocolo fornece meios para alterar o *timeout* facilmente. Entretanto, threads ou outros recursos costumam ser bloqueados enquanto o tempo limite não estoura ou a mensagem esperada não é recebida.

No entanto, o controle de tempo limite precisa ser feito manualmente quando não ocorre em consultas (até porque as consultas não são muito bem-vindas segundo a diretriz de Orientação a Mensagens assíncronas do Manifesto Reativo) ou quando protocolos assíncronos são usados.

Detectando tempo limite manualmente

Uma prática que resolve esse problema é a de persistir as transações pendentes de resposta e os respectivos timeouts no banco de dados local e, periodicamente, verificar se o seu prazo já foi esgotado (técnica conhecida como *polling*).

O agendamento de tarefas para o momento de expiração da mensagem é uma alternativa que pode poupar o *polling*. Em tais casos, é necessário considerar que o agendamento pode ser volátil e

será perdido se a aplicação for encerrada. Portanto, nesses cenários, a aplicação deve reagendá-las ao reiniciar.

Se a ferramenta da assincronia em questão possuir recursos, como atraso de mensagens, como é o caso do *sns/sqs* do *rabbitMQ* e do *KubeMQ*, essa tarefa é simplificada. Basta enviar uma mensagem para si mesmo, para ser entregue apenas após o tempo limite esgotado, indicando o fim da espera de uma transação.

As mensagens envolvidas devem ser idempotentes para que, se a transação já tiver sido efetuada ou cancelada, a mensagem atrasada não exerça influência no resultado. O contrário também deve ser gerenciado: se a mensagem atrasada chegar após o tempo limite, ela deve ser ignorada. A idempotência é facilmente alcançada ao usarmos o **identificador de relação**.

A vantagem do protocolo assíncrono em comparação ao síncrono, neste critério, é o não bloqueio de recursos e a consequente habilidade de permitir que outra thread/instância do serviço trate o esgotamento do tempo de espera. A desvantagem é, mais uma vez, a complexidade da implementação que não é tão trivial.

Configuração do *timeout*

Timeouts devem ser suficientemente altos para permitir que as respostas mais lentas cheguem, mas adequadamente baixos para encerrar uma espera de uma resposta que nunca vai chegar.

A contragosto de Legião Urbana e de Guns N' Roses, não temos todo o tempo do mundo e nem paciência. Calibrar o valor ideal é uma tarefa de tentativa e erro.

Uma piada interna no time em que eu trabalho diz que boa parte das perguntas da computação pode ser respondida com "começa com um valor qualquer e depois faz ajustes finos". Esse é o caso do valor ideal para o *timeout*.

Começa com, sei lá, três segundos e utiliza uma monitoria constante para ajustar o valor, seja acrescendo ou decrescendo. Falaremos mais sobre monitoria no capítulo *Observabilidade de sistemas*.

A próxima sequência mostra o `serviço-de-pagamento` aguardando a resposta do `serviço-antifraude` e abortando a operação após o tempo limite excedido.

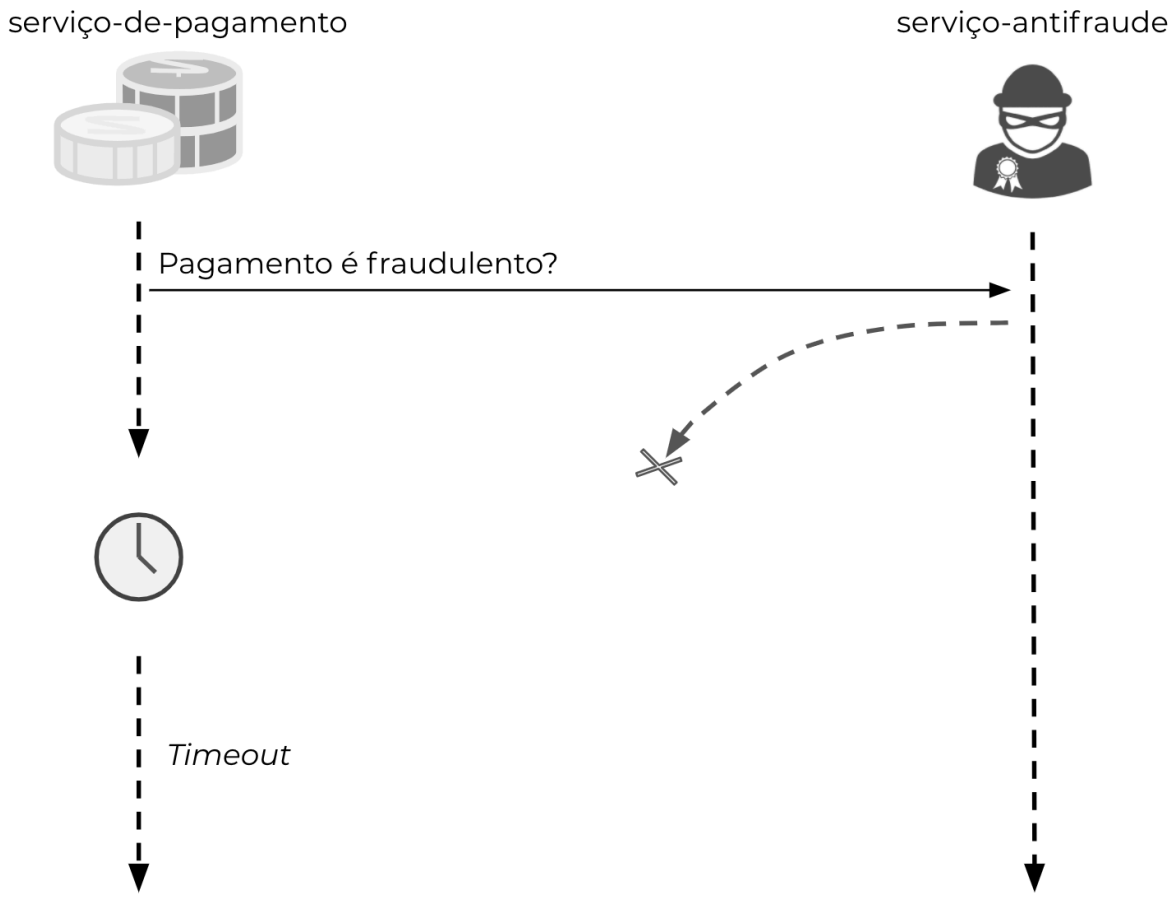

Figura 17.7: Exemplo de timeout.

Nos fluxos reativos, na ausência de mensagens síncronas, o padrão se torna uma ferramenta indispensável para tratamento de erros.

Solução alternativa (*Fallback*)

Este padrão é dedicado àqueles que, assim como eu, desfrutaram a infância na década de 90 e não conseguiram escapar da apaixonante família Hanson e sua balada romântica:

> *Won't you save me?*
>
> *Saving is what I need*
>
> – Hanson - *Save me.*

O padrão de *fallback* permite que uma tarefa continue a ser executada mesmo no caso de uma falha ou *timeout*. Em vez de repassar o insucesso adiante, usamos uma abordagem substituta. A abordagem estepe pode consistir em um valor padrão ou em uma consulta em um terceiro serviço.

A imagem a seguir mostra novamente o `serviço-de-pagamento` emitindo uma solicitação para o `serviço-antifraude`. Novamente, o `serviço-antifraude` retorna um erro de servidor. Desta vez, no entanto, após um *timeout*, o `serviço-de-pagamento` faz uso de um *fallback* e assume que a transação não é fraudulenta.

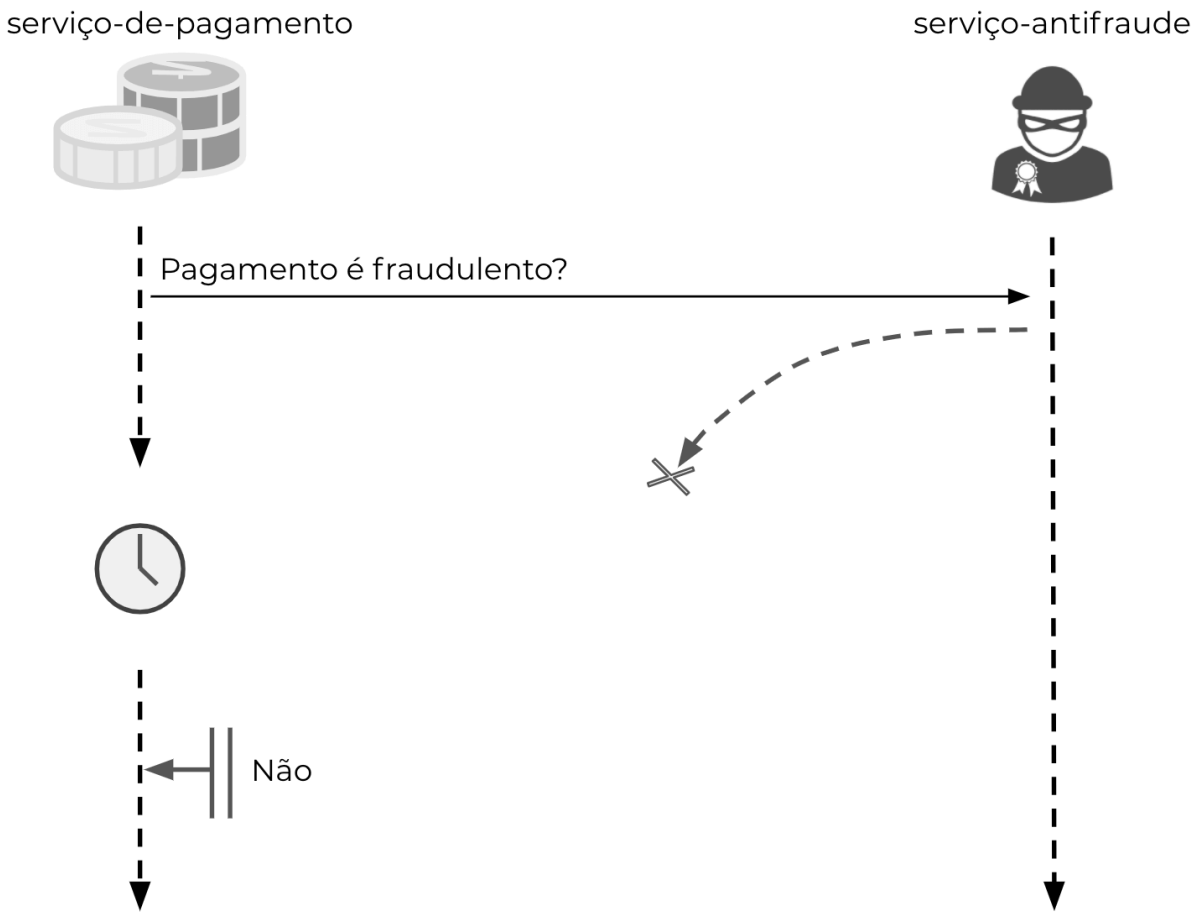

Figura 17.8: Exemplo de fallback.

Os valores de fallback nem sempre são possíveis, mas podem aumentar a resiliência se usados com cuidado.

No exemplo acima, pode ser perigoso tratar a transação automaticamente como não fraudulenta diante da indisponibilidade do `serviço-antifraude`. Essa lógica abre uma brecha de segurança e pode resultar em prejuízo financeiro. Usuários mal-intencionados podem tirar proveito da lógica e fraudar o sistema. Um exemplo de como fazer isso seria bombardear o `serviço-antifraude` ao ponto de torná-lo indisponível e, após isso, fazer transações fraudulentas que não seriam verificadas por conta da ausência do `serviço-antifraude`.

Por outro lado, se o *fallback* admitir que toda transação é fraudulenta, nenhum pagamento será realizado nos casos de falha

de comunicação com o `serviço-antifraude`.

Em tais cenários, uma boa lógica é recorrer a uma regra que não seja prejudicial para o negócio. Tal como condicionar o valor do *fallback* aos valores das transações.

Diante da indisponibilidade do `serviço-antifraude`, o `serviço-de-pagamento` entenderia todas as transações com valores maiores do que R$15,00, por exemplo, como fraudulentas e as demais seriam aceitas normalmente.

Um cálculo teria que ser feito para descobrir qual valor permitiria bom equilíbrio entre risco e perda de compras.

Tentar novamente (***Retry***)

O padrão *retry* é um oferecimento da inesquecível princesa do pop, Britney Spears e sua dançante interpretação de *Hit me baby one more time*.

> *When I'm not with you*
>
> *I lose my mind*
>
> *Give me a sign*
>
> *Hit me, baby, one more time*
>
> – Britney Spears - *Baby one more time.*

Figura 17.9: Opa, fiz isso de novo.

Sempre que assumirmos que uma resposta, ou um *timeout*, pode ser corrigida reenviando a mensagem, este padrão pode ajudar.

Ou seja, sempre que suspeitarmos de que o problema é uma falha transiente, a operação pode ser repetida, com intervalos entre as repetições, antes que a desistência seja oficialmente decretada.

A imagem a seguir mostra o `serviço-de-pagamento` tentando emitir uma verificação de fraude. A primeira solicitação falha devido a um

erro interno do `serviço-antifraude`. O `serviço-de-pagamento` tenta novamente e a resposta diz que a transação não é fraudulenta.

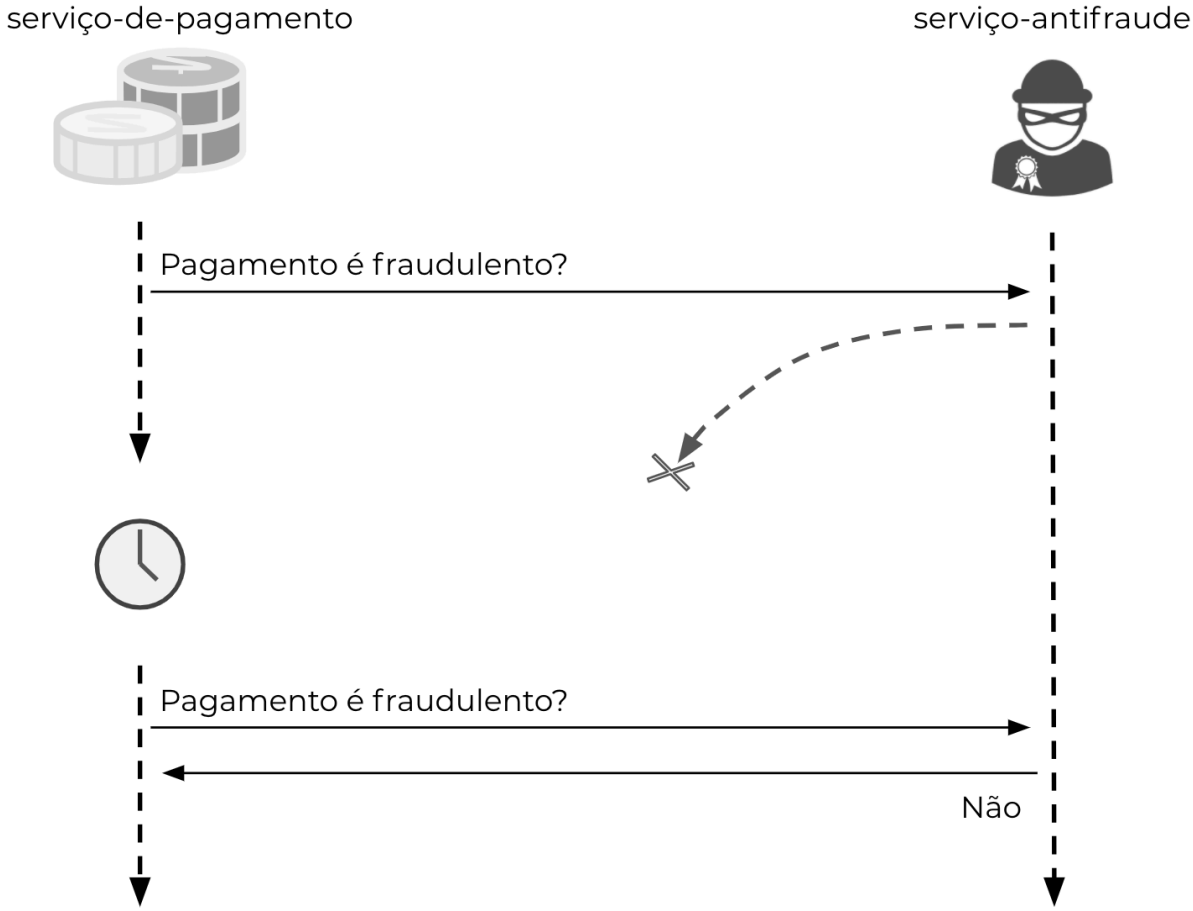

Figura 17.10: Exemplo de retry.

As mensagens em questão devem seguir as recomendações ACID 2.0 mencionadas nos capítulos anteriores. Caso contrário, as repetições poderão causar efeitos colaterais não intencionais.

Se o autor da chamada fizer novas tentativas, a tarefa poderá ser executada duas vezes, portanto devemos nos assegurar de que o processamento repetido não interfira no resultado.

Existem situações em que a sobrecarga do serviço pode ser configurada como uma falha transiente e tentar novamente pode agravar a situação. Se a falha ocorrer por conta de uma sobrecarga

no serviço, o que pode se configurar como uma falha transiente, tentar novamente sobrecarregará o serviço ainda mais.

Idealmente, a sobrecarga do serviço é sinalizada e resolvida antes que vire um problema maior, com mecanismos, como a **contrapressão** e o **processamento em lotes**, vistos na parte *Orientação a Mensagens*.

Ainda assim, caso a sobrecarga persista, repetir a chamada somente acentuará a situação e o serviço ficará ainda mais abarrotado. É nesse ponto que se esconde um segredinho especial do padrão.

Assim como quando juntamos a "pizza" e a "técnica de dobrar ao meio" nós criamos o calzone, o *retry*, quando equipado com intervalos gradualmente maiores entre as chamadas, vira ***exponential backoff***.

Em vez de fazer uma nova chamada imediatamente após o erro e piorar a sobrecarga, aumenta-se o tempo de espera entre as repetições após cada insucesso.

Dessa forma, a aplicação sobrecarregada terá mais tempo de colocar o trabalho em dia e, na próxima requisição, responder conforme o esperado.

Por exemplo, quando a solicitação falha na primeira vez, tentamos novamente após um segundo. Se falhar pela segunda vez, esperamos dois segundos antes da próxima tentativa. Se a segunda tentativa falhar, esperamos quatro segundos antes da próxima.

Portanto, estamos dobrando o intervalo entre as solicitações após cada falha. O tempo de espera deve depender do aplicativo e do serviço e pode seguir qualquer lógica, como: aumentos exponenciais, Fibonacci, fixos, lineares e qualquer outra que sua criatividade for capaz de conceber.

O padrão se alia ao *timeout* e ao *fallback* de modo que: 1) o *timeout* indica a falha; 2) usa o *backoff exponencial* para solucionar o

problema; e 3) adota o *fallback* como solução.

Se percebermos que a solução tem falhado consistentemente, podemos nos precaver e, temporariamente, adotar o *fallback* de cara, sem sequer tentarmos nos comunicar com o destinatário original. Dessa forma, introduzimos um novo padrão: *circuit breaker*.

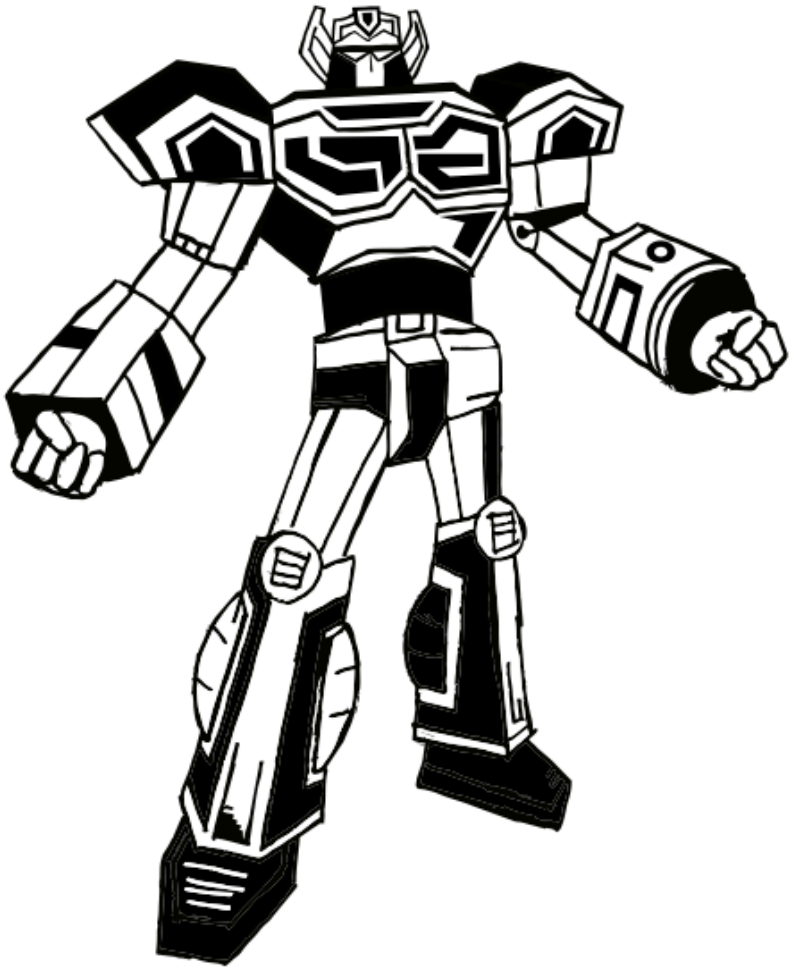

Figura 17.11: Timeout + exponential backoff + fallback.

Disjuntor (*Circuit breaker*)

Circuit breaker é o nosso rockstar da turma e ele nada mais é do que a extensão e a combinação dos padrões anteriores.

Na eletrônica, um disjuntor é uma chave que protege seus componentes contra danos por sobrecarga. No software, um disjuntor protege seus serviços de solicitações inoportunas, embora a solicitação seja parcialmente efetuada.

> *Cause you're hot then you're cold*
>
> *You're yes then you're no*
>
> *You're in then you're out*
>
> *You're up then you're down*
>
> – Kate Perry - *Hot n'cold.*

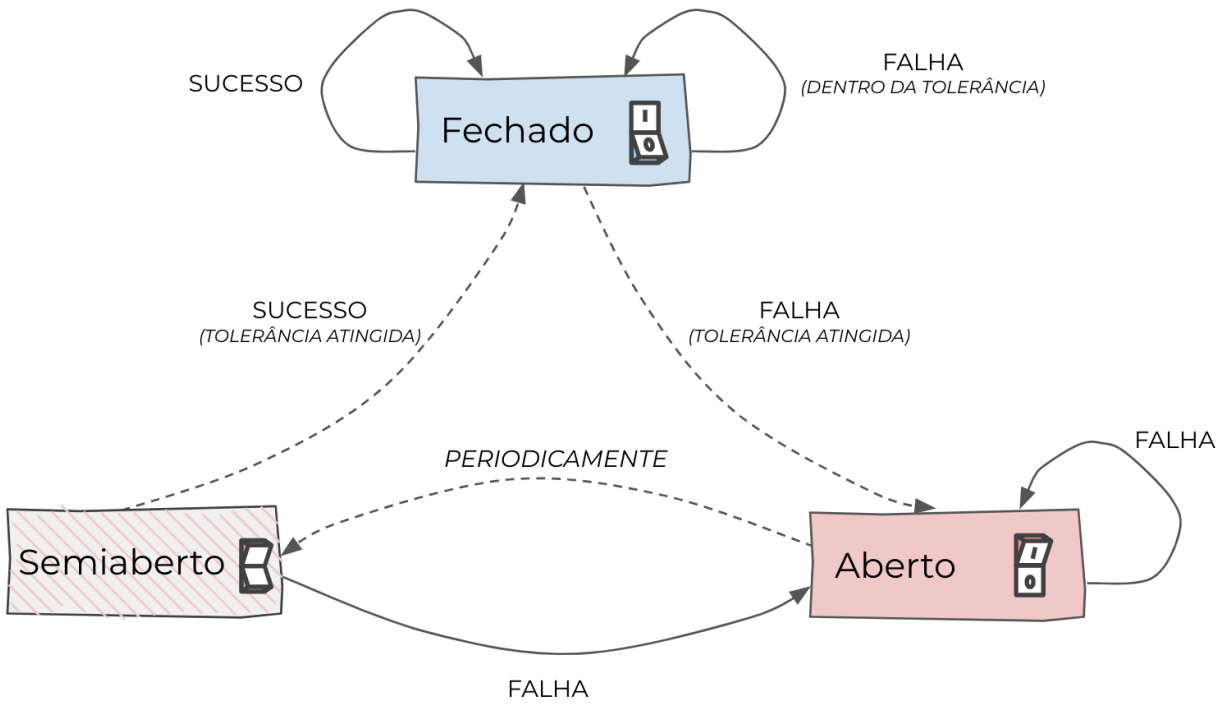

Figura 17.12: Estados de um disjuntor de software.

A ideia básica por trás do disjuntor é muito simples. Você envolve uma operação em um *circuit breaker* para monitorar suas falhas.

Uma vez que as falhas atingem um certo limite, o disjuntor desarma, dispara um alarme para monitoria, e as chamadas adicionais para o disjuntor sofrem curto-circuito e retornam um valor desejado pré-configurado, sem sequer atingir a operação protegida.

O padrão do disjuntor foi descrito por Martin Fowler e popularizado por Michael T. Nygard no livro *Release It!: Design and Deploy Production-Ready Software*. Link do artigo de Martin Fowler: https://martinfowler.com/bliki/CircuitBreaker.html.

O padrão pode ser implementado como um componente de software com um modo de operação que alterna entre três estados. Os autoexplicativos **fechado** e **aberto** e um que fica entre os dois, meio lá inclinado para cá, o **semiaberto**.

Figura 17.13: Meio lá inclinado para cá.

- **Estado fechado:** as solicitações fluem livremente. Tudo está funcionando bem e todas as chamadas passam para os serviços remotos. Enquanto as falhas são esporádicas, o disjuntor entende que não é necessário abrir o circuito. Após uma determinada quantidade de insucessos, o disjuntor dispara, e o estado muda para aberto.

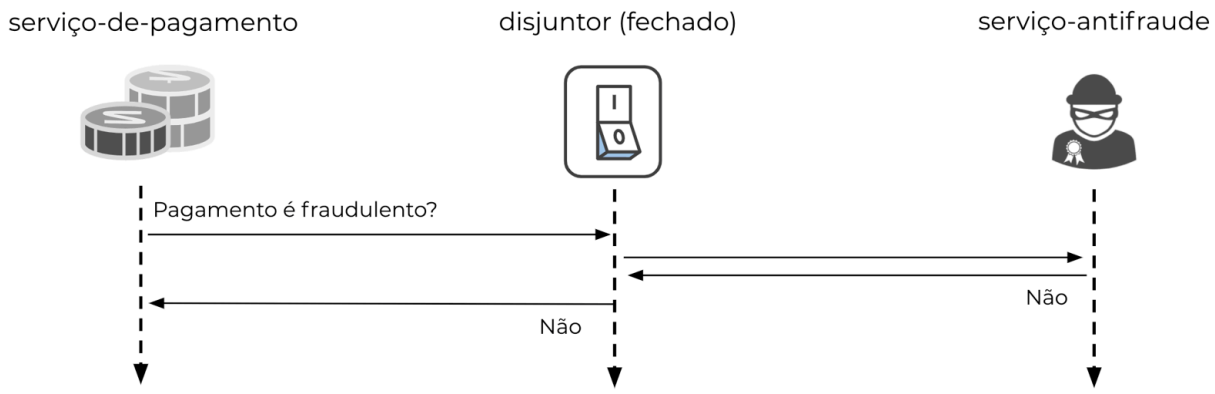

Figura 17.14: Disjuntor fechado.

- **Estado aberto**: as solicitações são rejeitadas sem sequer serem enviadas ao recurso remoto, evitando um agravamento de sobrecarga e preciosos segundos de espera. Nesse estado, o disjuntor pode retornar erro ou um valor predeterminado. Após um valor configurável de tempo, o disjuntor muda para o estado semiaberto automaticamente.

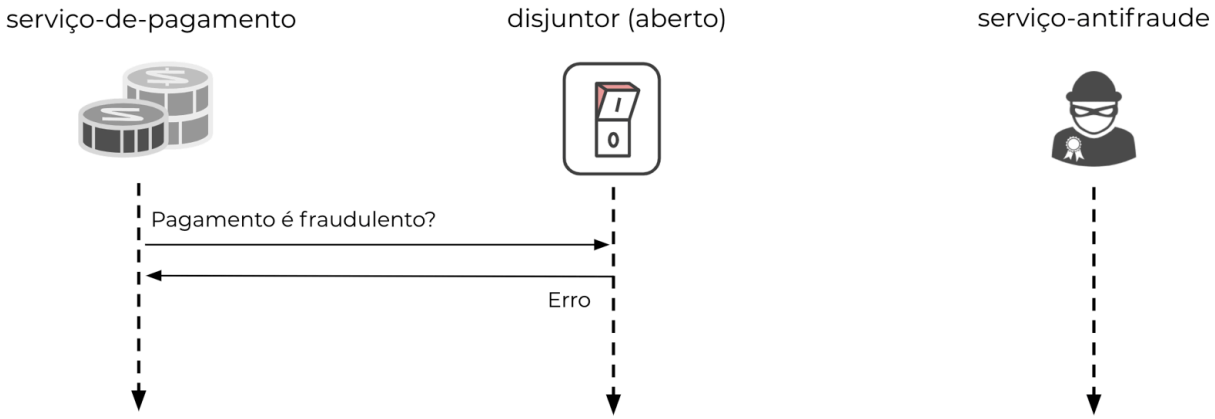

Figura 17.15: Disjuntor aberto.

- **Estado semiaberto**: uma pequena quantidade de solicitações de sondagem decide se fecham o circuito e voltam ao ideal ou se retornam ao estado aberto. Normalmente as solicitações de sondagem são determinadas periodicamente, por exemplo, uma sondagem a cada dez segundos; ou em relação à quantidade

original, por exemplo, uma solicitação a cada cem solicitações com curto-circuito.

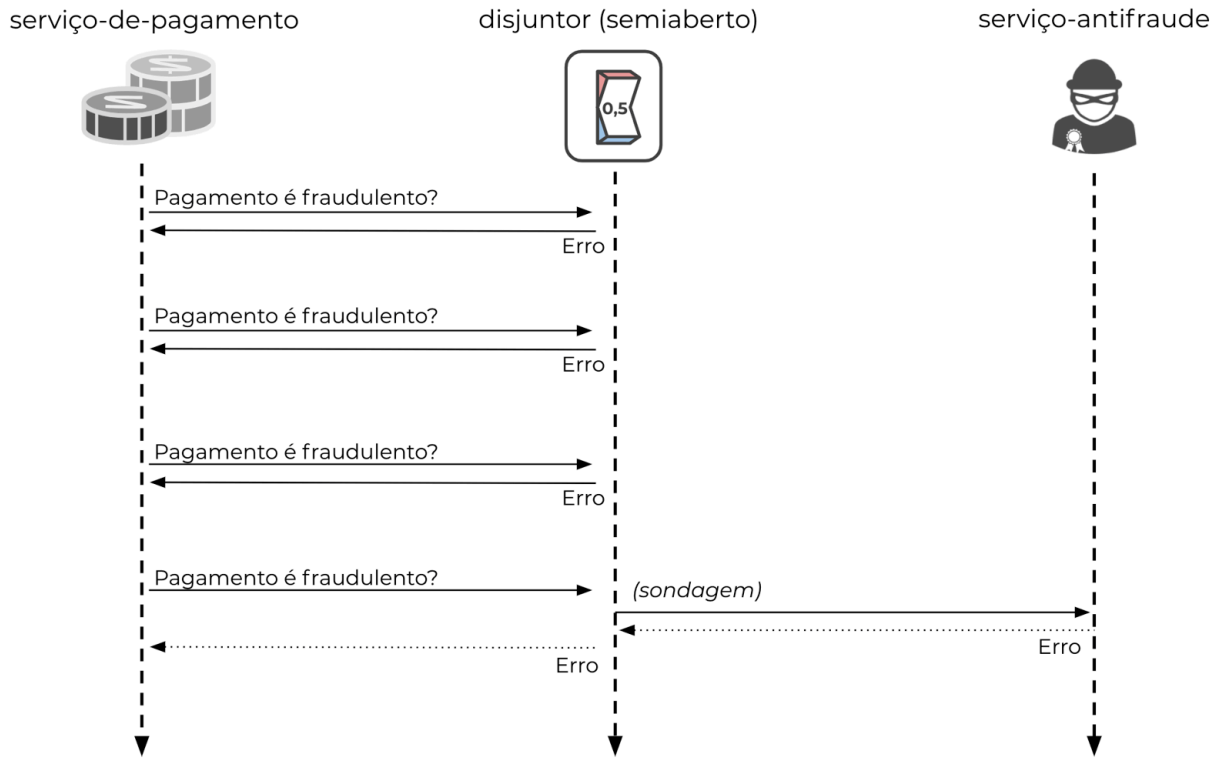

Figura 17.16: Disjuntor semiaberto.

Considere o seguinte exemplo no cenário em que estamos trabalhando. A solicitação do `serviço-de-pagamento` para o `serviço-antifraude` é passada através do disjuntor. Depois de um número configurável de erros do servidor, o circuito é aberto e as solicitações subsequentes são bloqueadas.

Após um tempo parametrizável de espera, o circuito entra no estado semiaberto. Nesse estado, ele permitirá que uma quantidade de solicitações seja aprovada com o propósito de sondar e decidirá, de acordo com o resultado dessas solicitações, se o circuito deve continuar aberto em caso de falha, ou se será fechado em caso de sucesso.

Por conta própria, os disjuntores ajudam a reduzir os recursos bloqueados em operações sujeitas a falhas, inclusive em fluxos

assíncronos e reativos.

Por exemplo, o disjuntor pode abrir o circuito quando uma fila estiver cheia, ou quando as operações recentes do sistema estiverem estourando o tempo limite, ou quando a mensagem de contrapressão indicar um esgotamento iminente do sistema.

Você evita esperar pelo *timeout*, e um circuito interrompido evita adicionar carga em um sistema com problemas.

Embora geralmente se fale de chamadas remotas, caso comum de disjuntores, elas podem ser usadas em qualquer situação em que se deseje tanto proteger partes de um sistema passível de falha quanto economizar possíveis bloqueios de recursos. Os disjuntores também são locais muito valiosos para monitoramento. Sempre que o estado do disjuntor mudar, ele deve ser registrado e revelar detalhes de seu estado para um monitoramento mais aprofundado.

O comportamento do disjuntor é muitas vezes uma boa fonte de avisos sobre problemas mais graves no ambiente. A equipe de operações também deve ser capaz de desarmar ou reinicializar os disjuntores.

Configuração do disjuntor

Os disjuntores fornecem um abundante leque de parametrizações. Além do tipo de erro, a quantidade e a frequência de cada tipo de erro podem ser parametrizados para configurar o disparo do disjuntor.

Assim como também podem ser valores configuráveis, o intervalo, a frequência, a quantidade e a definição de sucesso nas solicitações de sondagem para fechar o circuito novamente.

17.4 A resiliência corrói a robustez

Como bem pontua Uwe Friedrichsen, em uma palestra intitulada de *The 7 Quests of Resilient Software Design* no GOTO 2018, os padrões de resiliência, resumindo com muita coerência:

1. São opcionais.
2. Aumentam a complexidade, portanto, corroem a robustez.
3. Possuem custo associado em devops.
4. Possuem orçamento limitado.
5. Escolhem os padrões complementares.

A palestra está disponível por meio do link: https://www.youtube.com/watch?v=v8hh0mB35wQ.

O ponto número 2 chama bastante atenção. Ao passo que os *fallbacks* e *circuit breakers* crescem em quantidade, mais difícil se torna testar todas as combinações possíveis de fluxo, o que pode resultar em soluções não testadas e obsoletas em ambiente de produção.

Como são soluções usadas em cenário de erro, a ausência de testes pode criar uma falha pior do que a original, que eles deveriam resolver. Portanto, paradoxalmente, quanto mais complexa é a solução de resiliência, que visa melhorar a robustez, menos robusta ela é.

Para evitar tais catástrofes, uma solução possível é testar constante e automaticamente essas alternativas. Por exemplo, para uma pequena parcela do fluxo em produção, abrir o disjuntor mesmo sem haver erro e verificar antecipadamente se o *fallback* funciona, antes que seja necessário usá-lo.

Outra maneira de antecipar testes de resiliência é deliberadamente encerrar serviços funcionais e ver como o sistema reage a isso. Soa como suicídio assistido, mas a Netflix batizou essa técnica como *Chaos Monkey*.

17.5 Conclusão

> *É só isso*
>
> *Não tem mais jeito*
>
> *Acabou, boa sorte*
>
> – Vanessa Da Mata - *Boa sorte.*

Ao som de baladas inesquecíveis, vimos que o padrão de *timeout* fornece um limite superior para a latência. O *fallback* ajuda a resolver falhas de comunicação localmente. O padrão de *retry* permite lidar com erros de comunicação que podem ser corrigidos em várias tentativas. Os disjuntores agem proativamente e evitam que algumas chamadas sejam feitas por assumir que a operação desejada ainda está apresentando irregularidades.

Figura 17.17: Hits pops.

Resiliência é isso. É abraçar o caos do mundo real, porque você não pode controlá-lo e traduzir essa forma de pensar em arquiteturas de software.

A arte de gerenciar sistemas em escala está em abraçar a falha e estar no limite - empurrando os limites do desempenho do seu sistema e software quase ao ponto de ruptura, mas ainda assim sendo capaz de se recuperar.

Os sistemas resilientes removem componentes falhos da solução e os reintegra quando voltam a ser operacionais. Eles aceitam a ideia de que errar faz parte.

Em sistemas de grande escala, a probabilidade de 100% de excelência operacional é quase impossível de ser alcançada.

Portanto, o estado normal de operação é uma falha parcial.

Embora não seja adequado para todo tipo de aplicação, a execução em modo de falha parcial é uma opção viável para a maioria. Os padrões de resiliência são adequados para uma indústria em que a demanda é constante e não controlada e em que transações individuais podem ser sacrificadas sem perdas catastróficas.

Embora o cenário exemplificado mostre o funcionamento dos padrões em consultas síncronas, eles também são aplicáveis em comunicações assíncronas.

No geral, pode-se afirmar que, nos sistemas reativos, os padrões de resiliência são mandatórios. Embora aprender como projetar e implementar padrões de resiliência seja relativamente fácil, os desafios reais não estão no domínio da codificação e sim em saber quais e quando usar.

Padrões de resiliência compõem um assunto extenso, e poderia muito bem haver livros com maior dedicação a eles.

Roland Kuhn, um dos autores do Manifesto Reativo, Roberto Vitillo e Robert S. Hanmer pensaram o mesmo e destinaram valiosas páginas dos seus livros, ou os próprios livros inteiramente, a esse objetivo. Na ordem: *Reactive Design Patterns*, *Understanding Distributed Systems* e *Patterns for Fault Tolerant Software*.

Nesses livros, você encontrará mais detalhes sobre os padrões aqui descritos e conhecerão alguns não citados aqui, como *Fail-fast*, *Bulkhead*, *Health Endpoint Monitoring*, *Fail over*, *Load Balancing*, *Leader Election*, *Scheduler Agent Supervisor*, *Service mesh* e muitos outros.

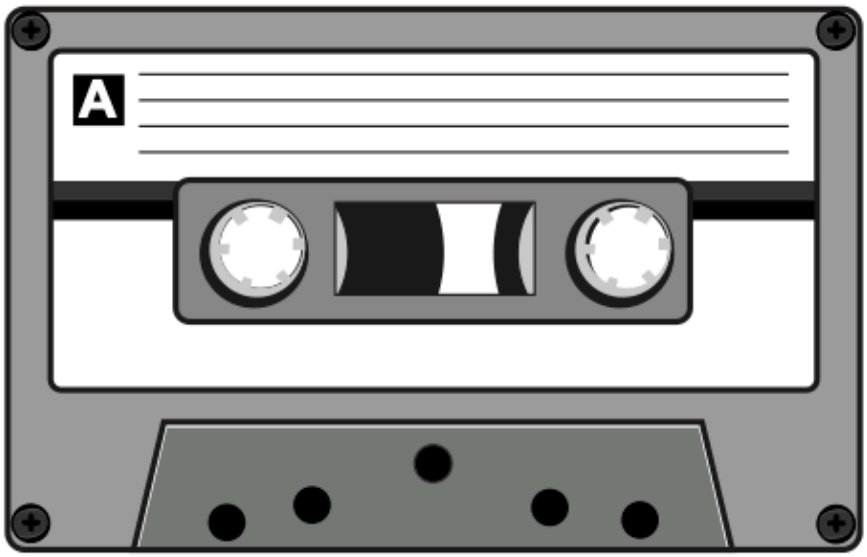

Figura 17.18: Interessados na playlist, procurem no Spotify: Virgs - Baladas reativas.

Capítulo 18
Transações distribuídas

Figura 18.1: Pobre Saurinho.

Como temos visto, as transações entre serviços são sujeitas a falhas e o ideal é minimizá-las. Todavia, as transações ignoram qualquer vontade e teimam em existir de qualquer forma, afinal os serviços precisam de algum meio para interagir e trocar informações entre si.

Não obstante, muitas vezes, um fluxo exige mais do que uma, duas ou três transações com relações causais. Repletos de rotas alternativas e bifurcações, os fluxos podem se tornar um verdadeiro emaranhado que dariam inveja a qualquer espaguete.

Para exemplificar o que quero dizer, vou contar uma história quase original.

18.1 Uma jornada inesperada

Como bem sabem os amantes da literatura fantástica tolkieniana, Sauron, o Senhor das Trevas, é um empreendedor de sucesso na Terra Média. Além de abrigar e oferecer emprego aos menos favorecidos e esquecidos pela sociedade, Sauron também possui negócios rentáveis nas indústrias da agricultura, bélica, de joias e hoteleira.

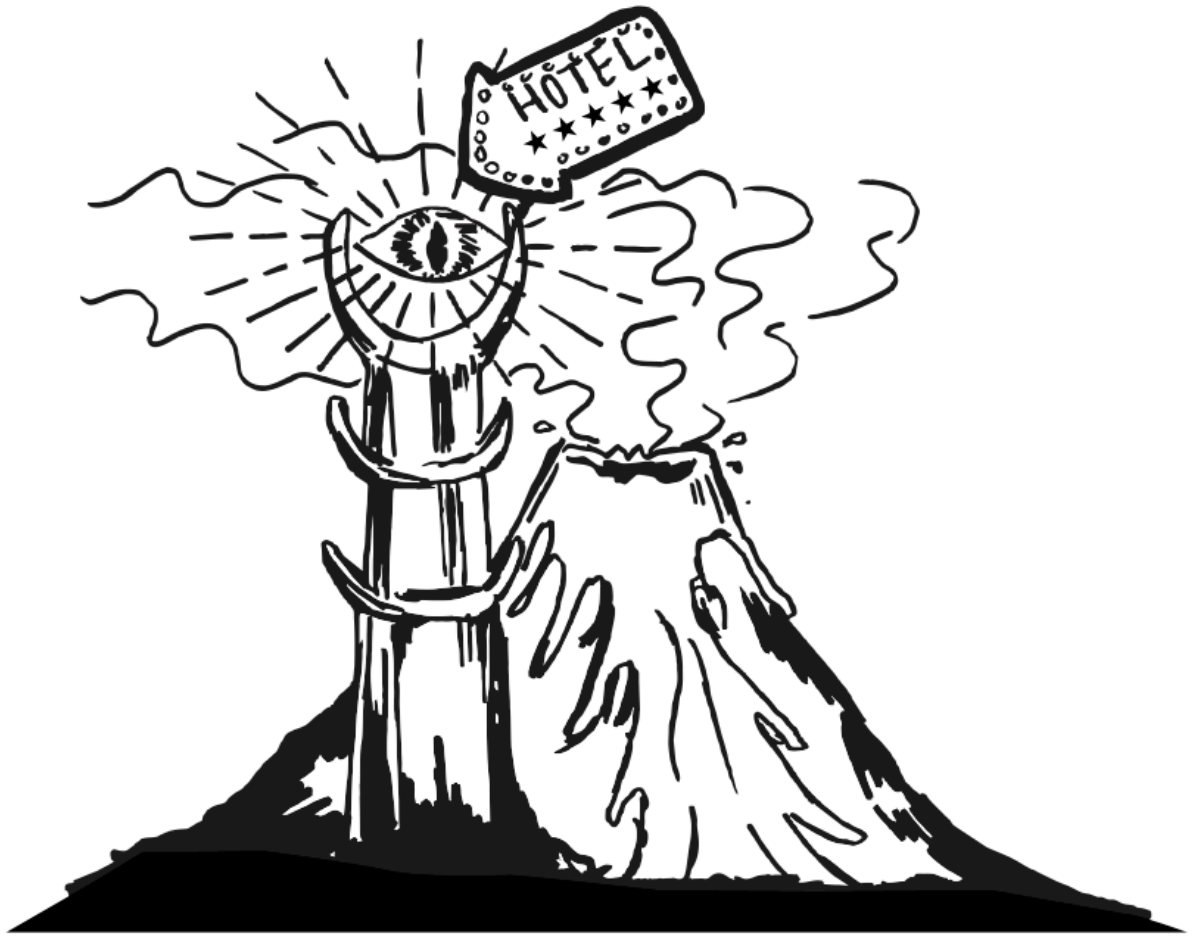

Figura 18.2: Empreendimento de sucesso do Sauron.

Para a infelicidade de Sauron, malfeitores invejosos roubaram, junto com sua dignidade, um bem de valor inestimável, o anel da família.

Revoltado com a ineficiência dos tiras, ele não mede esforços para recuperar o seu maior bem e decide buscá-lo por conta própria.

Por não possuir um corpo físico e, portanto, ser incapacitado de ir a pé ao covil do inimigo, o Condado, o carismático justiceiro opta por ir de avião. E, por não possuir mais o vigor de outrora, o Senhor das Trevas precisará se hospedar lá por perto antes do retorno triunfal.

O roteiro da viagem contempla uma passagem de avião de ida e volta para o Condado, a cidade do ladrão, e uma diária na pousada

Pônei Saltitante. Se der tudo certo, Sauron voltará a tempo para assistir à sua novela imperdível.

Para a sorte do aventureiro, há um *e-commerce* magicamente elaborado para dar conta de todo o plano de viagem. Sua arquitetura engenhosa de microsserviços é contemplada por, entre outros, um microsserviço encarregado da reserva do voo, outro para agendar o quarto do hotel e outro para realizar pagamentos.

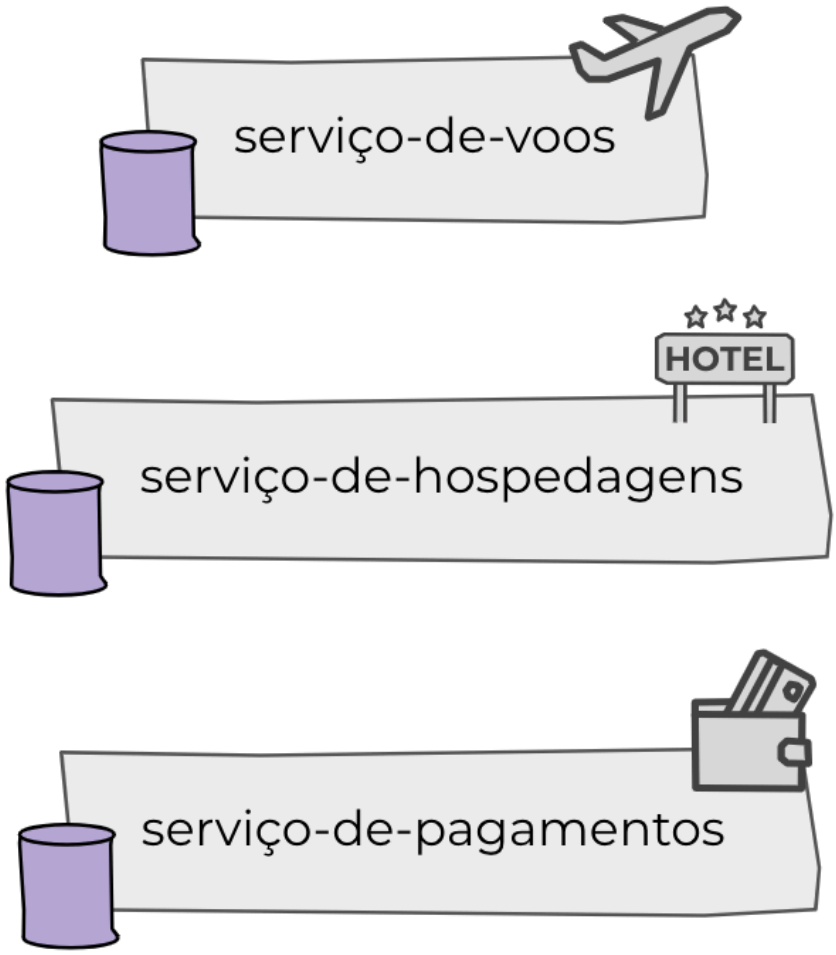

Figura 18.3: Arquitetura de serviços do e-commerce.

Cada microsserviço sabe como desempenhar sua tarefa ou reverter a operação caso seja necessário. O `serviço-de-pagamentos` sabe como

confirmar ou cancelar o pagamento e fazer o estorno, o `serviço-de-hospedagens` tem conhecimento de como reservar e desocupar os quartos, e o `serviço-de-voos` entende como agendar e anular passagens aéreas.

Um requisito indispensável para o usuário, no caso o Sauron, é que todos os serviços devem realizar suas operações e, se alguma delas falhar, a solução deve garantir que todas as operações anteriores sejam invalidadas. O que habilitaria o protagonista da história a pensar em uma solução alternativa.

Por exemplo, não seria de muita utilidade, sem contar o desprazer, manter o voo agendado e o quarto reservado quando o cartão de crédito não é aceito pelo *e-commerce*. Todas as operações devem ser canceladas.

Nós já vimos esse filme antes, sabemos que o *e-commerce* precisa garantir que a transação que envolve todas essas etapas seja **atômica**.

Outra garantia que a abordagem precisa dar é do nível de **isolamento**. Seria no mínimo constrangedor se, após a confirmação do pagamento, a última etapa da transação, a vítima da história fosse impedida de concluir a operação porque outro passageiro pegou o assento inicialmente reservado para o justiceiro.

Em uma solução monolítica, que possui um único banco de dados, isso seria facilmente assegurado fazendo uso das propriedades mágicas ACID dos SGBD relacionais usuais. Contudo, esse não é o caso, pois, como o limite geral da transação cruza a fronteira de vários serviços e, por conta do princípio de Repositório Único, a transação é considerada uma **transação distribuída**, nenhuma das conveniências ACID locais é útil nesse contexto.

18.2 Confirmação de duas fases (2PC)

A confirmação de duas fases (do inglês *two phase commit*, ou apenas 2PC) é um mecanismo bem conhecido para implementar uma transação em sistemas de banco de dados e alcançar os benefícios ACID.

Esse mecanismo também pode ser usado para microsserviços para implementar transações distribuídas. Um participante que costuma dar as caras para a efetivação da 2PC é o `serviço-gerenciador`, microsserviço que atua como organizador da transação.

A 2PC funciona em duas fases:

1. Fase de preparação:

O `serviço-gerenciador` percorre todos os serviços participantes da transação consultando a possibilidade de que a transação ocorra, pedindo para que eles se **preparem para a operação**.

Cada componente retorna uma resposta indicando se a operação poderá ser concluída com êxito. O `serviço-gerenciador` espera pela resposta de todos os participantes e, se todas forem positivas, ele dá seguimento à 2PC.

Se um dos serviços se negar, o gerenciador aborta a operação e comunica a decisão para todos os envolvidos.

É como se o gerenciador mandasse um "prepara, que agora é a hora" para que todos os envolvidos ficassem de prontidão e reservassem os recursos dos quais são encarregados.

Nessa fase, **não há modificação permanente de estados**, é só um aquecimento para a próxima fase.

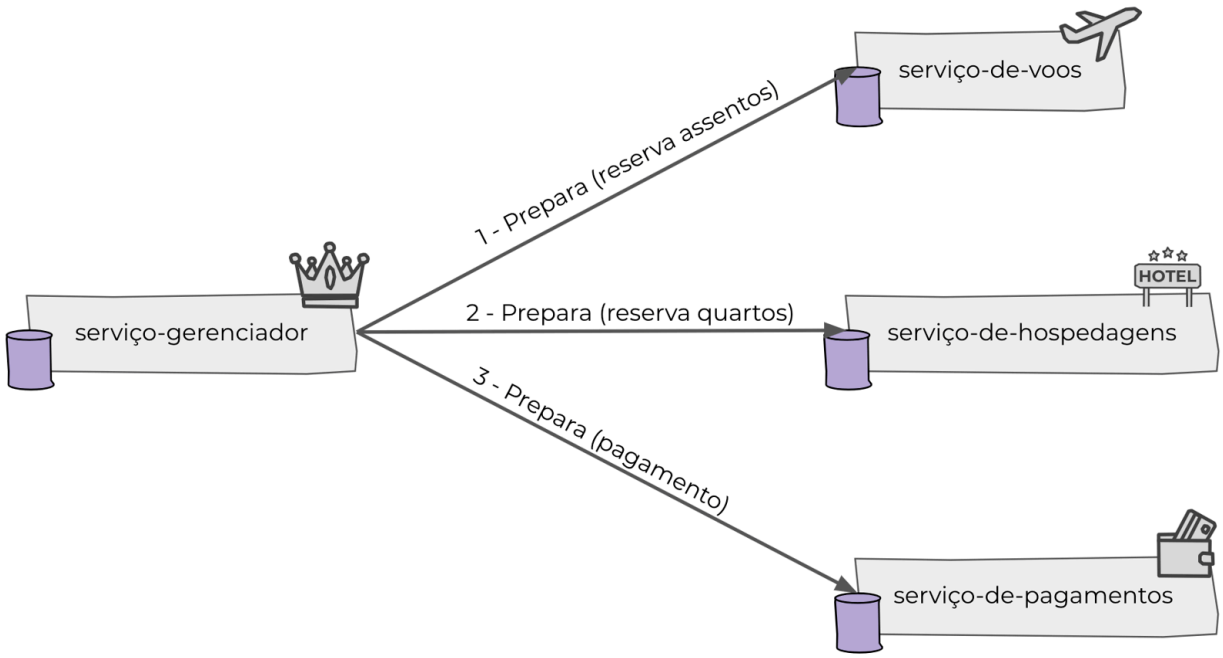

Figura 18.4: Fase de preparação da 2PC.

2. Fase de confirmação:

Uma vez que todos concordam com a solicitação de preparação e reservam os recursos, o `serviço-gerenciador` perpassa, sequencialmente, o mesmo caminho anterior **confirmando a operação**. Ponto em que, finalmente, há modificações de estado.

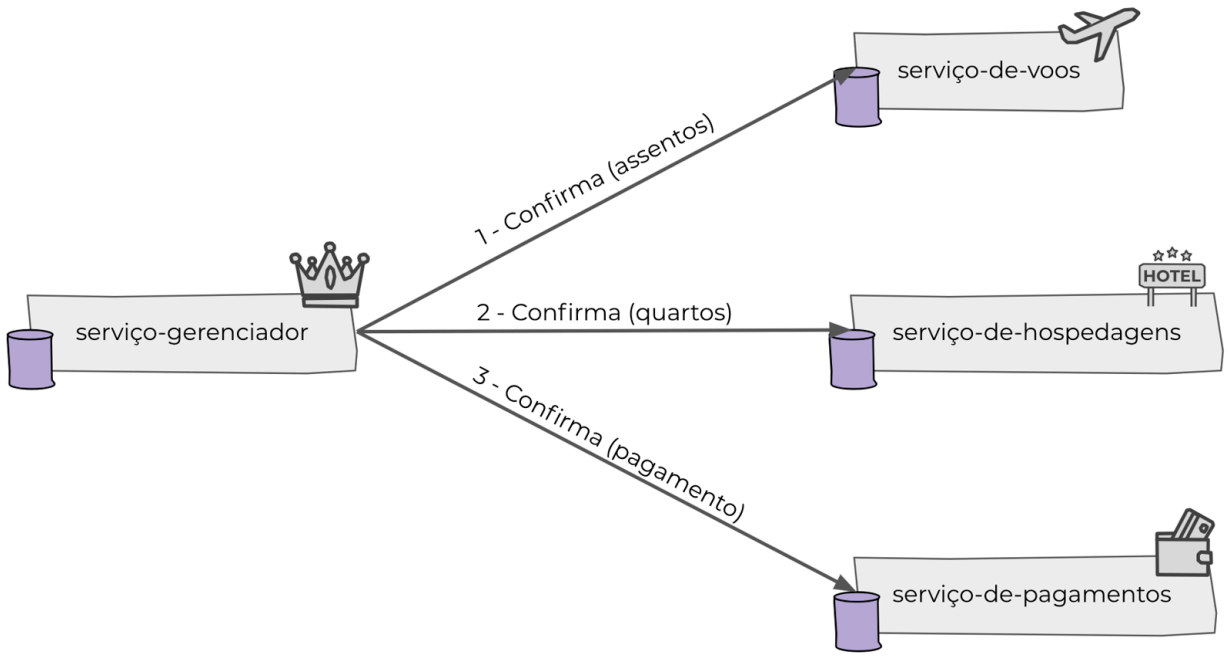

Figura 18.5: Sucesso na fase de confirmação da 2PC.

A 2PC resolve o contratempo do isolamento e da atomicidade. Uma vez que todos os serviços responderam positivamente à fase de preparação, nenhuma outra operação pode afetar os dados envolvidos, garantindo o isolamento.

A atomicidade é garantida pela fase de confirmação. Se for bem-sucedida, a transação ocorre com sucesso. Caso contrário, todas as operações confirmadas devem ser revisitadas e revertidas.

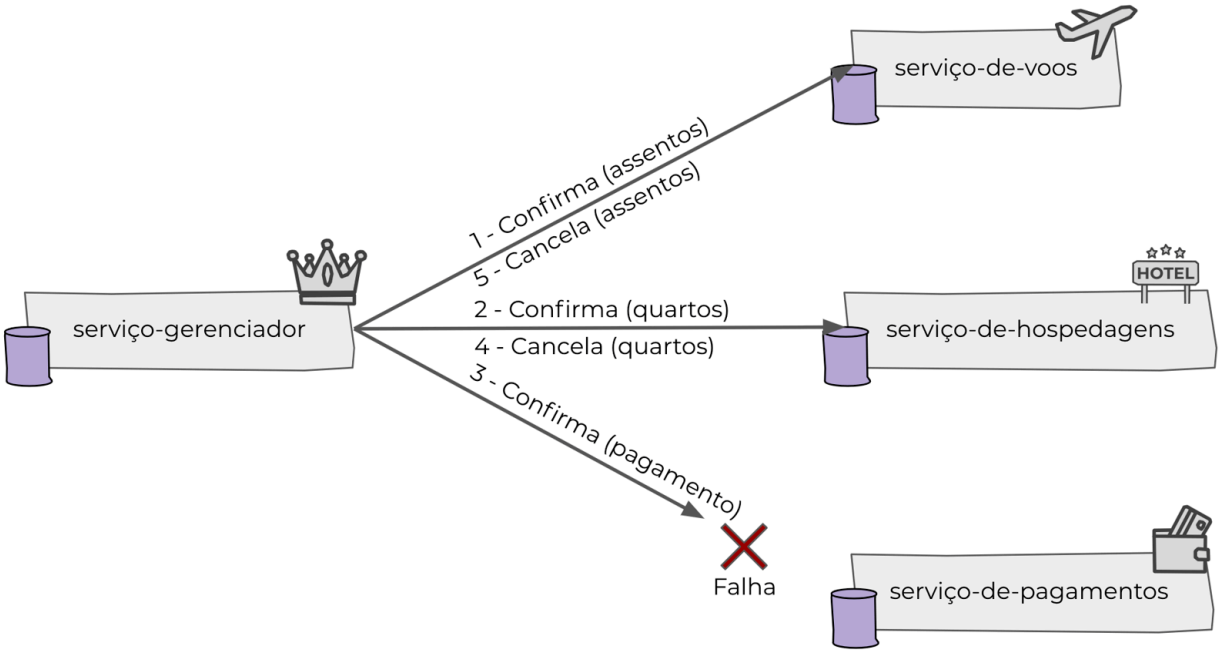

Figura 18.6: Falha na fase de confirmação da 2PC.

Embora a 2PC possa resolver as pendências, ela também se torna o único ponto de falha, pois o ônus da transação recai sobre os ombros do gerenciador.

Ponto único de falha

Um ponto único de falha (SPOF, do inglês *Single Point Of Failure*) é um risco potencial de que uma falha em um componente acometa toda a arquitetura. Se algum imprevisto ocorrer e o ponto em questão ficar indisponível, a transação será afetada e paralisará.

A resiliência necessária para superar o SPOF exige um custo para empregar um mecanismo de alta disponibilidade, por exemplo, o preço de servidores adicionais dentro de um *cluster*.

Em ambientes menores, isso pode funcionar bem, mas as coisas desmoronam quando você está falando sobre centenas de microsserviços. Em alguns casos, o custo para contornar um SPOF é maior do que os benefícios da operação em risco.

Um excelente exemplo é o menino Neymar na copa do Mundo de 2018. Se for um microsserviço que fica bastante indisponível ou que caia muito, como o Neymar, a seleção e o povo brasileiro sofrem as consequências.

Outro ponto fraco do mecanismo é o desempenho geral. Como cada serviço é percorrido duas vezes e o `serviço-gerenciador` é sobrecarregado, toda a transação é limitada pelos recursos mais lentos, já que qualquer serviço pronto tem que esperar pela confirmação de outro mais lento.

A despeito do protocolo de comunicação adotado, as fases são de **natureza bloqueantes**. Após a execução da primeira e antes do término da segunda, todos os recursos envolvidos não podem ser alterados. De outro modo, o isolamento da solução seria comprometido.

Se a segunda fase demorar muito ou mais do que o previsto, a solução reduz o rendimento e pode causar adversidades na

escalabilidade, especialmente em um cenário envolvendo muitos serviços.

Sabemos que os sistemas reativos não combinam com operações bloqueantes e, como se não bastasse esse revés, um desempenho ruim frustraria o roteiro traçado por Sauron. Nós não queremos frustrar os planos do Sauron, não é?

Fora os já citados, outro grande problema dessa abordagem é que não há mecanismo para reverter a outra transação se um microsserviço ficar indisponível na fase de confirmação. Ou seja, se passou da fase de preparação, tem que ir até o final. Não existe o conceito de voltar atrás.

É aí que entram SAGA e as operações de compensação.

18.3 SAGA

SAGA é, em sua essência, um padrão de gerenciamento de falhas. Ela surge da constatação de que transações de longa duração ou muito distribuídas não podem ser facilmente manipuladas usando o modelo de confirmação de duas fases.

Em vez disso, o padrão SAGA divide o trabalho em transações individuais cujos efeitos podem ser, de alguma forma, revertidos após o trabalho ter sido executado e confirmado.

Conforme descrito por Hector Garcia-Molina e Kenneth Salem em seu artigo intitulado SAGAS (disponível no link http://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf), da *Association for Computing Machinery*, SAGA é um padrão que envolve uma sequência de operações locais que realizam uma unidade de trabalho específica.

Cada serviço envolvido em uma SAGA realiza sua própria transação e publica um evento. Os outros serviços ouvem esse evento e executam a próxima transação local publicando um novo evento escutado pelo microsserviço seguinte e assim sucessivamente.

A essência da ideia é que uma transação distribuída de longa duração pode ser vista como a composição de várias etapas transacionais locais rápidas.

Cada etapa é emparelhada com uma ação de reversão compensatória, reversão em termos da semântica do negócio, não necessariamente voltando o estado do componente ao que era antes da ação original, de modo que toda transação distribuída possa ser revertida em caso de falha executando a ação compensatória de cada etapa. Idealmente, essas etapas devem ser comutativas para que possam ser executadas em paralelo.

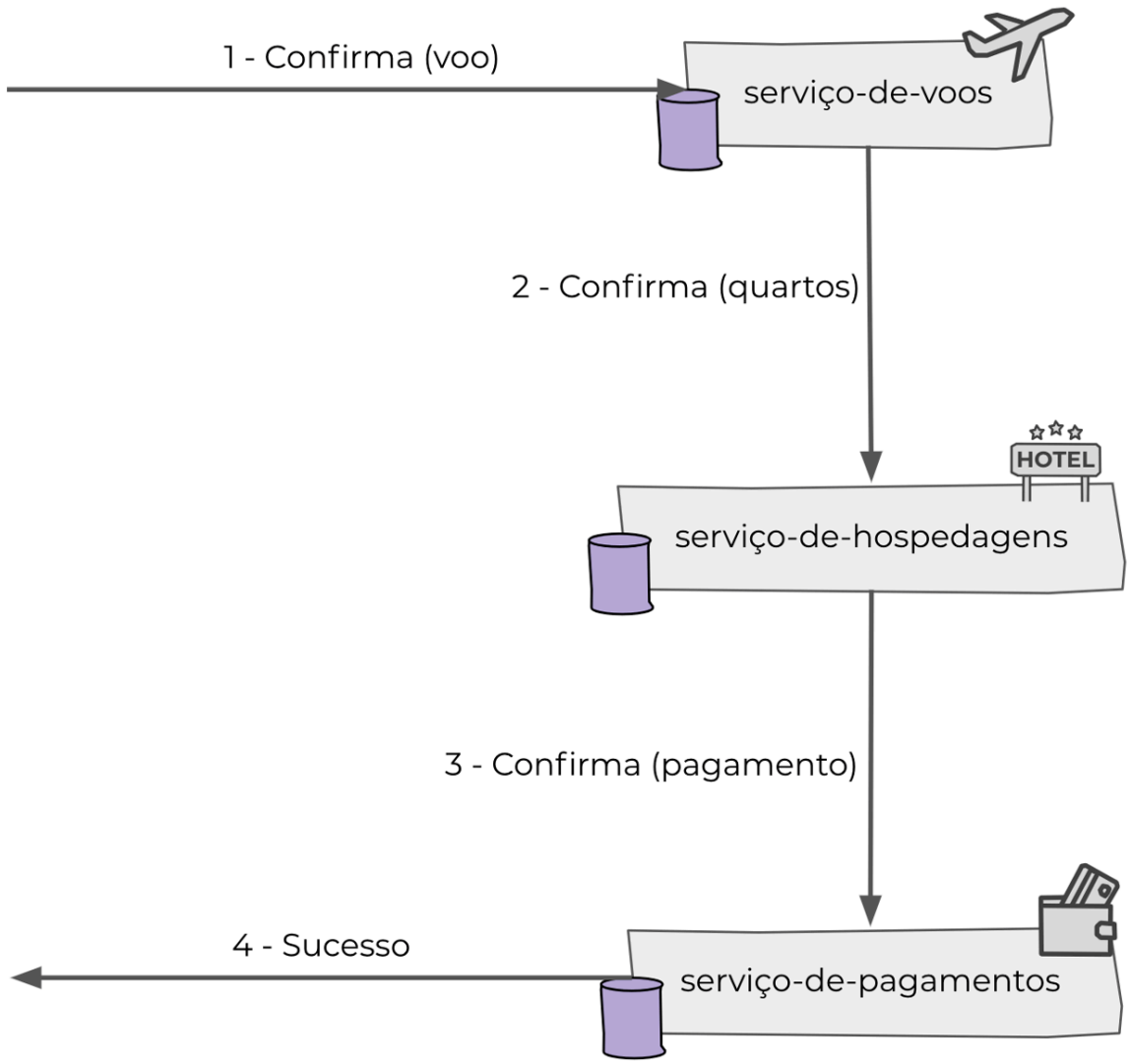

Figura 18.7: Transação SAGA bem-sucedida.

A transação de compensação não precisa ser exatamente a reversão da original, *ipsis litteris*. Por exemplo, não precisa apagar a compra da história e negar qualquer assunto sobre ela, basta dizer que ela foi cancelada.

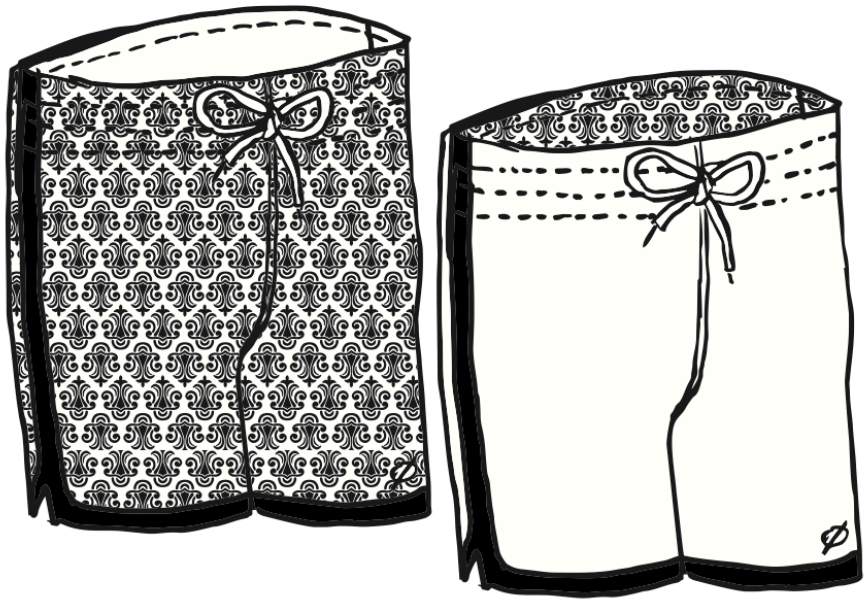

Figura 18.8: Sistemas reativos, não confundir com sistemas reversivos.

No caso de falha em um serviço no fluxo de trabalho, o padrão reverte a transação geral por meio de mensagens de compensação encadeadas em ordem cronológica reversa de cada microsserviço atuante.

Para isso, a partir do serviço que falhou, o padrão exige que o componente faça a limpeza de seus recursos e envie uma **mensagem de compensação** para o serviço executado logo antes.

Isso move a execução para a etapa anterior, que executa ações de compensação para reverter as alterações feitas por sua transação local, e repete a operação de contatar seu serviço anterior para as operações de compensação dele.

Por exemplo, se após a confirmação do voo e dos quartos o `serviço-de-pagamento` acusar erro ao processar a operação, o `serviço-de-hospedagens`, que é o serviço que notou a falha, deve desfazer sua própria operação, a reserva de quartos, e transmitir a falha para o

serviço anterior, o `serviço-de-voos`, no caso. A operação é repassada até chegar ao serviço que originou o fluxo.

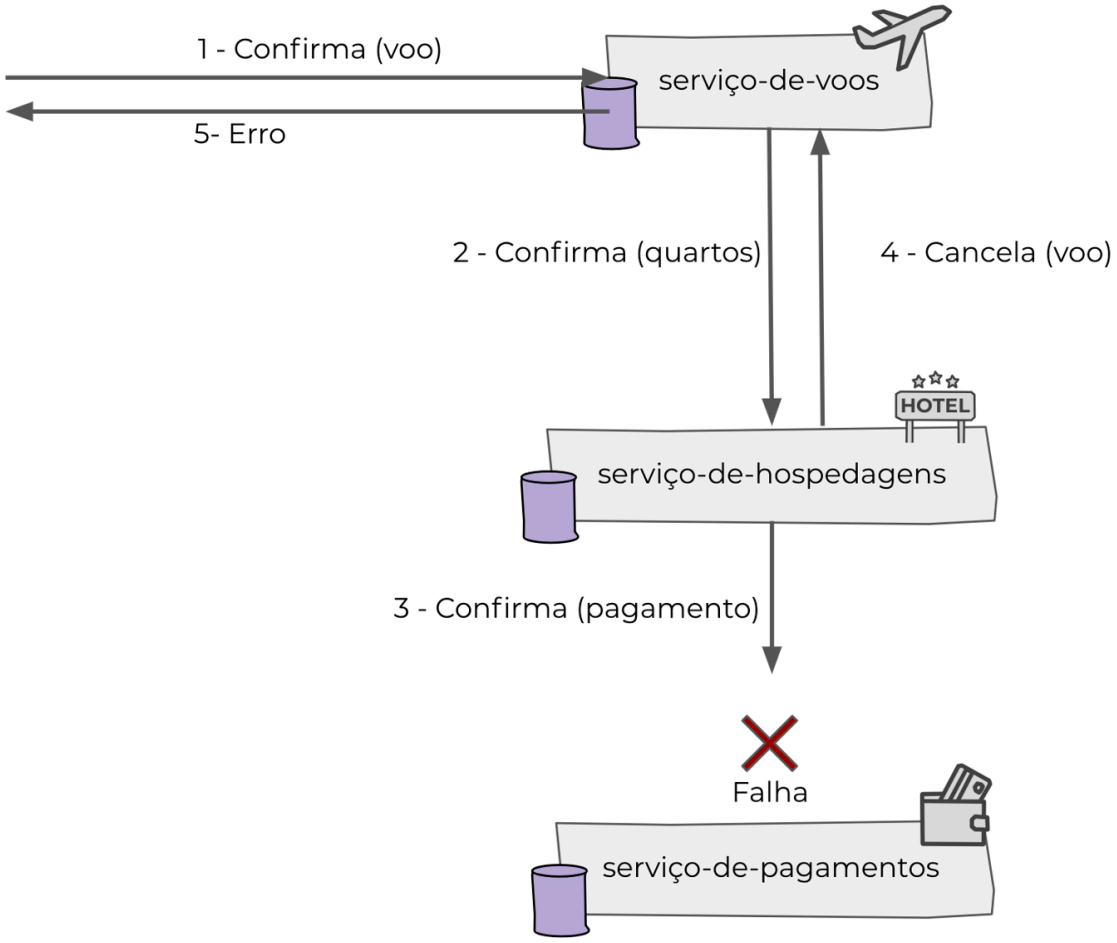

Figura 18.9: Transação SAGA com falha e transação de compensação.

Propriedades das mensagens do padrão SAGA

Das propriedades ACID 2.0, uma transação de compensação deve ser idempotente e comutativa, ou seja, deve possuir a capacidade de ser repetida e lidar com a reordenação, já que a transação de compensação pode chegar antes da mensagem que a originou ser completamente processada.

Embora o padrão ajude a manter a consistência dos dados em vários serviços sem um acoplamento forte, ele não garante a consistência

forte e introduz um certo nível de complexidade de um ponto de programação. O plano de implementação deve tratar as transações de compensação como prioridade.

Em relação à 2PC, o SAGA melhora a arquitetura geral por enfraquecer o acoplamento geral enquanto enrijece o acoplamento entre o microsserviço e seus vizinhos. O padrão SAGA tem um melhor desempenho do que a 2PC e não oferece nenhum ponto único de falha. Além disso, o padrão também garante a consistência causal dos dados.

Dois fatores distinguem as metodologias. O primeiro é a fase inicial de preparação necessária somente na 2PC, o segundo é a existência da transação de compensação presente somente na SAGA, e a consequente exigência de que cada microsserviço envolvido saiba como executar e reverter a operação pela qual é responsável.

18.4 Conclusão

No mundo de microsserviços, em que cada microsserviço tem seu próprio armazenamento de dados, manter a consistência dos dados é difícil. Os microsserviços introduziram um conjunto de problemas para o gerenciamento de transações, pois cada um dos serviços orientados ao domínio é implementado individualmente e executado de forma isolada. Com uma arquitetura de microsserviços, um único processo de negócios reúne vários microsserviços para fornecer uma solução geral, inviabilizando por completo transações ACID, em alguns casos.

A 2PC e a SAGA são dois modelos propostos que ajudam a solucionar o problema. Sauron poderá então ter sua joia em mãos e se dedicar ao mundo dos negócios.

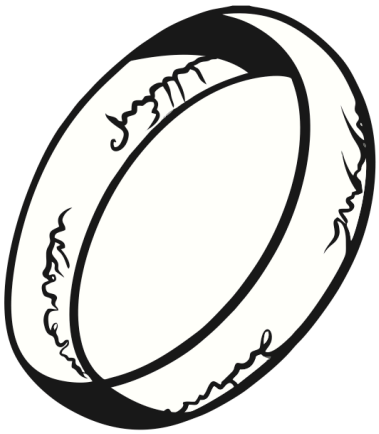

Figura 18.10: Joia encontrada com sucesso.

Ambas as metodologias podem ser implementadas com protocolos de comunicação de qualquer estrutura que não mudarão sua essência. Além disso, os dois modelos podem ser implementados com a utilização de um serviço gerenciador de transação, que é mandatório no caso da 2PC. O gerenciador pode ser incorporado a algum microsserviço ou, na maioria dos cenários, pode ser um componente autônomo.

A 2PC fornece um protocolo de consistência forte para lidar com transações. No entanto, essa abordagem geralmente não é recomendada para microsserviços por vários motivos. Em primeiro lugar, ela tenta gerenciar o requisito de atomicidade bloqueando cada recurso durante a fase de preparação da transação.

Enquanto garante a consistência, o bloqueio limita a disponibilidade do serviço, excluindo o acesso de outros chamadores durante a transação. No pior cenário, duas transações podem potencialmente bloquear os recursos uma da outra, causando a ocorrência de um *deadlock* distribuído.

Por último, além dos problemas causados pelo bloqueio de recursos, as características de desempenho de um gerenciador de transações distribuídas diferem significativamente de um gerenciador não distribuído. As transações distribuídas estão sujeitas a problemas

adicionais de conectividade de rede e problemas de latência de comunicação. Além disso, a confirmação de duas fases é síncrona por natureza. O gerenciador de transações deve aguardar uma resposta de cada recurso participante antes de decidir o destino da transação. E já sabemos que as operações síncronas afetam negativamente a disponibilidade e geralmente são evitadas com arquiteturas de microsserviço reativas.

Como vimos, a SAGA é uma ótima ferramenta para garantir a atomicidade em transações de longa duração. É importante entender, todavia, que isso não fornece uma solução para o isolamento. SAGAS executadas simultaneamente podem afetar umas às outras e causar erros. A depender do caso de uso, isso não pode ser aceitável, então você precisaria adotar uma estratégia diferente, como garantir que a SAGA não ultrapasse vários limites de consistência ou simplesmente usar um mecanismo diferente para o trabalho.

Comparando uma abordagem em relação a outra, vimos que 2PC é mais fácil de ser implementada, mas possui pior desempenho geral. SAGA, no que lhe diz respeito, é mais complexa de ser implementada e demanda as transações de compensação. Por consequência, a aplicação do mecanismo exige uma mentalidade mais avançadas para as equipes de desenvolvimento.

Apesar de geralmente acompanhada de um `serviço-gerenciador`, a 2PC pode ser implementada sem a presença de tal. De modo análogo, há uma variação do padrão SAGA que apresenta um serviço coordenador de transações.

No próximo capítulo, entenderemos que a existência de um `serviço-gerenciador` caracteriza um modelo de coordenação de fluxos muito comum: a orquestração. De modo contrário, a ausência dessa figura em um fluxo de dados representa a coordenação coreografada. Também examinaremos mais a fundo outras diferenças entre os modos de coordenação.

Capítulo 19
Coordenação de fluxos

Neste capítulo, veremos que existem duas abordagens para coordenar o fluxo de mensagens e fazer os microsserviços trabalharem em conjunto em direção a um objetivo comum: **orquestração** e **coreografia**.

Essas abordagens identificam o modo de coordenação de um fluxo de mensagem e não o próprio microsserviço, o que quer dizer que o mesmo serviço, como parte integrante de vários fluxos, pode ser componente de orquestrações e de coreografias ao mesmo tempo.

A orquestração envolve controlar ativamente todos os elementos e interações como um maestro dirige os músicos de uma orquestra, enquanto a coreografia envolve estabelecer um padrão ou rotina que os microsserviços seguem enquanto a música toca, sem exigir supervisão e instruções.

Falaremos sobre esses diferentes espetáculos da indústria de entretenimento em detalhes a seguir.

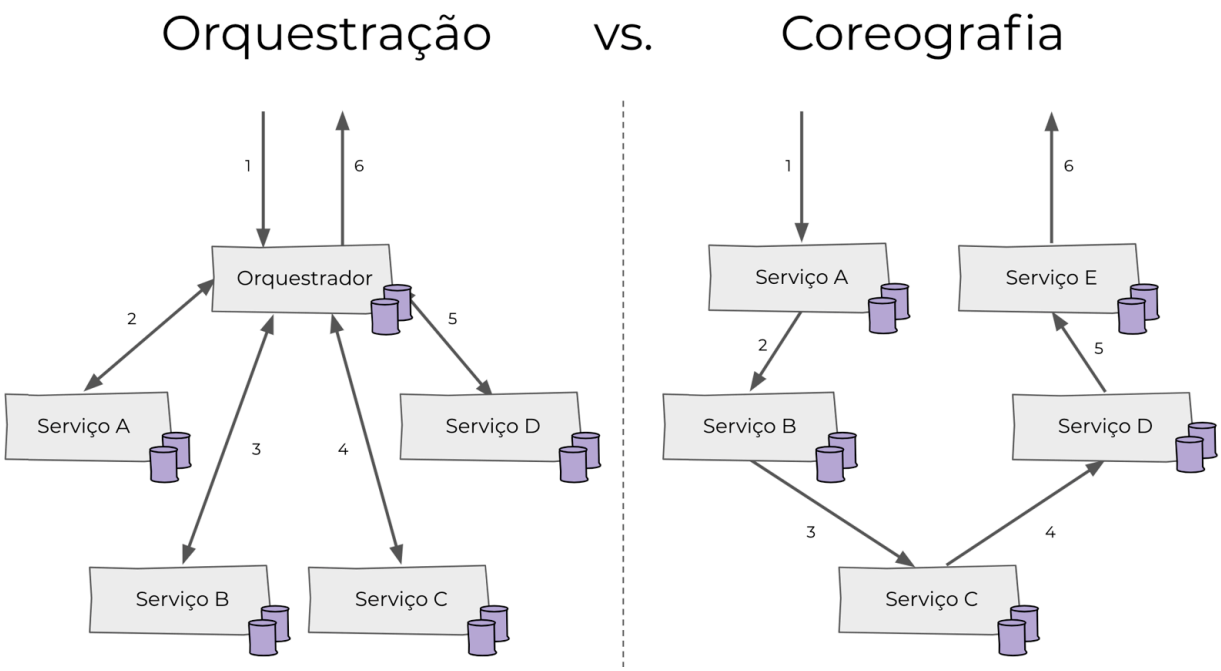

Figura 19.1: Orquestração vs. coreografia.

19.1 Orquestração

Em uma orquestra, os músicos atendem aos comandos do maestro. Cada um deles é um especialista em tocar seu instrumento e mesmo possuindo uma instrução particular, a partitura, estará coletivamente perdido sem um orientador para guiá-lo.

Em um fluxo orquestrado, um controlador lida com todas as comunicações entre os microsserviços e direciona cada serviço para executar a função pretendida. Enquanto na sinfonia a função pretendida é "tocar uma baita música coletivamente", na orquestração, um exemplo de função seria "confirmar uma compra".

A orquestração emprega uma abordagem centralizada para a composição de serviço. Você reúne vários serviços por uma lógica fixa descrita em um único processo de negócios executável centralizado, o orquestrador, que é responsável por controlar as

regras de negócio, invocar sucessivas chamadas aos serviços e garantir a consistência e a taxa de transferência do fluxo.

O orquestrador transmite as mensagens e responde a elas. Ele chama um serviço e só invoca o próximo serviço quando recebe a resposta do anterior.

Isso segue um paradigma de requisição e resposta, o que não é, necessariamente, síncrono. Enquanto não são tocados pela batuta do maestro orquestrador, os serviços aguardam pacientemente pela sua vez.

O padrão do orquestrador reduz a comunicação ponto a ponto entre os serviços, mas acopla fortemente o orquestrador a outros serviços que participam da transação. Para executar tarefas em sequência, o orquestrador precisa ter todo o conhecimento de domínio sobre as responsabilidades desses serviços.

Você pode imaginar a orquestração como um séquito de minions, do filme *Meu Malvado Favorito*. Diferentes indivíduos, asseclas incapazes de pensamento crítico, súditos do vilão com princípios nefastos, o Gru.

Os minions, como os serviços na orquestração, obedecem ao controlador sem ousar questionar seus pensamentos. Isso ressalta um ponto fraco da orquestração. Caso o orquestrador seja incompetente e/ou limitado, toda a arquitetura sofre esses efeitos.

Figura 19.2: Os Beatles estão diferentes.

As chamadas entre o orquestrador e os demais serviços permitem modelar o processo transacional como uma máquina de estado. Cada uma das etapas concluídas representa um estado. A máquina de estado deve ser persistida no banco de dados do orquestrador para se recuperar de quaisquer falhas no fluxo de orquestração.

19.2 Coreografia

A coreografia se apresenta como uma revolução ao aplicar um golpe na ditadura imposta pelo orquestrador e distribuir suas atribuições entre os diversos serviços participantes.

Figura 19.3: Sistemas reativos, não confundir com sistemas revolutivos.

Geralmente implementada com uma arquitetura Orientada a Mensagens, a coreografia deixa cada serviço decidir quando e como uma operação é processada, em vez de depender de um orquestrador central. Cada componente do sistema participa do processo de tomada de decisão sobre o fluxo e não possui um ponto central de controle.

Os microsserviços coreografados são serviços funcionando em harmonia, sem hierarquia, totalmente independentes, que, individualmente, não têm controle do fluxo geral das mensagens.

Na coreografia, cada serviço está incumbido apenas de seu próprio trabalho a partir das mensagens que recebe dos outros. Depois que

uma mensagem inicial é enviada para um serviço, ele a processa e gera a próxima mensagem para o serviço seguinte.

Não há dependências rígidas entre eles, portanto, diferentemente do orquestrador, são fracamente acoplados. A abordagem tem uma pegada de **arquitetura reativa**: os serviços sabem quando e como reagir quando recebem uma mensagem, porque essa lógica é inserida no microsserviço com antecedência. Em resumo, cada um deles se preocupa apenas com:

1. As mensagens em que ele está interessado.
2. O que deve fazer.
3. Quais mensagens ele deve emitir quando concluir sua tarefa.

Você pode imaginar a coreografia como um espetáculo do *Ballet Bolshoi*, maior referência de dança clássica do mundo, ou como os Vingadores – que nada mais são do que o *Bolshoi* com uma boa computação gráfica.

Assim como nos microsserviços, os bailarinos da companhia se agrupam e suavemente desfilam sobre o palco com as ferramentas que lhes pertencem: flexibilidade e graciosidade, ou, no caso dos microsserviços, a lógica e sua pilha tecnológica. No espetáculo e na coreografia, ninguém possui a lógica de decisão e liderança e todos são capazes de desempenhar seu papel sem a orientação de ninguém.

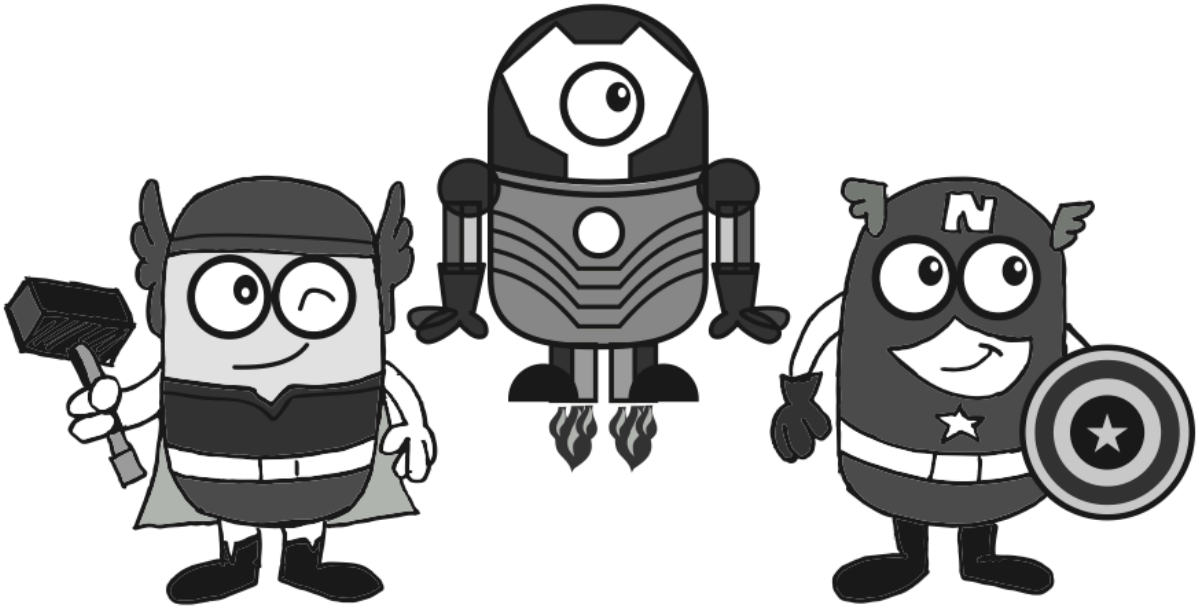

Figura 19.4: Banana vs. Banana.

Cada serviço não é responsável apenas pela resiliência de sua operação, mas também pela resiliência de todo o fluxo. Essa responsabilidade pode ser um fardo muito pesado para o serviço e difícil de implementar.

Os serviços devem tratar as falhas transitórias, não transitórias e de tempo limite, para que a solicitação seja encerrada normalmente, se necessário. Além disso, devem ser diligentes em comunicar o sucesso ou o fracasso da operação para que os demais serviços possam agir em conformidade.

Para não perder a continuidade do fluxo de trabalho, um serviço deve confirmar a leitura da mensagem somente após a transação do banco de dados ser realizada e a próxima mensagem ter sido disparada para o serviço seguinte.

Esse fluxo garante que não se perca nenhuma mensagem e a transação geral seja atômica: concluída com êxito ou que todas as operações sejam revertidas.

O surgimento das arquiteturas *FaaS* (*Function as a Service*, em português, função como serviço), inerentemente Orientadas a Mensagens, tornou a abordagem da coreografia muito popular. Esse padrão é um modelo natural para a arquitetura sem servidor, onde todos os serviços podem ter vida curta. São ativados devido a um evento, realizam suas tarefas e são removidos após a conclusão.

19.3 Orquestração vs. coreografia

Tanto a coreografia quanto a orquestração são maneiras de gerenciar as interações e o comportamento entre seus microsserviços.

Muitas vezes, não existirá uma abordagem que sirva para todos os casos e ter escolhido uma delas em uma solução anterior não significa que seja a correta para a próxima. Mais uma vez, trata-se de um *trade off*.

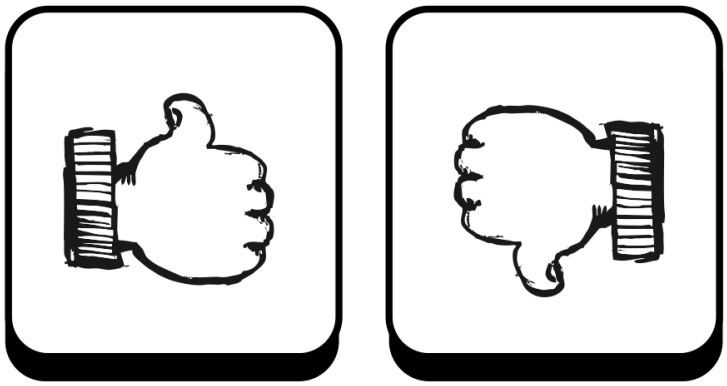

Figura 19.5: Prós e contras.

Vantagens da coreografia sobre a orquestração

Escalabilidade

A coreografia não é limitada pela escalabilidade de um único serviço. Cada etapa do fluxo pode ser alterada e dimensionada de forma independente.

Desempenho

A orquestração incorre em uma sobrecarga de desempenho devido à necessidade do orquestrador de se comunicar diretamente com cada serviço e aguardar a resposta.

A sede por controle e a tagarelice do orquestrador limitam o desempenho inteiro do fluxo à sua própria performance, que já é sobrecarregada por definição.

Não há ponto único de falha

O fato de a coreografia isolar os microsserviços envolvidos significa que, se um aplicativo falhar, os demais serviços podem continuar enquanto o problema é corrigido. Também não é necessário que cada serviço tenha tratamento de erros complexo e integrado no caso de falhas. Alguns serviços simplesmente não possuem tanta importância e aceitam falhas.

Na orquestração, caso o orquestrador se mostre indisponível momentaneamente, todos os fluxos em andamento e os que estão por vir são afetados. O mesmo aconteceria em uma sinfonia se o maestro perdesse a capacidade de administrá-la com eficácia.

No imortal programa televisivo do Chaves, há um exemplo perfeito do que ocorre quando o maestro não consegue dar conta do recado. No episódio, devido à inabilidade da maestrina, a Chiquinha, cada indivíduo toca uma música diferente e a orquestra da vila descamba no fiasco total. A exceção, claro, fica por conta do Quico e sua lacrimogênea interpretação da Dança das Horas, de Tchaikovski.

Figura 19.6: Se você é jovem ainda, jovem ainda, jovem ainda.

Custo

Manter o risco de um ponto único de falha é caro. Ademais, no paradigma de requisição e resposta exigido pela orquestração, mais mensagens são transitadas do que no paradigma utilizado pela coreografia, ou seja, mais latência e mais custo.

Modificabilidade

Microsserviços coreografados permitem que as equipes de desenvolvimento operem com maior independência e se concentrem em seus principais serviços. Depois que esses serviços foram criados, eles agora podem ser facilmente compartilhados entre as equipes.

Essa reutilização de componentes desenvolvidos, além de agilizar o desenvolvimento, é uma grande economia de trabalho e tempo.

Acoplamento fraco

Adicionar e remover serviços é muito mais simples em uma arquitetura de microsserviços coreografada. Tudo o que você precisa fazer é conectar o microsserviço (ou desconectá-lo) ao canal apropriado.

Com o acoplamento fraco, a adição e remoção de microsserviços não quebra a lógica existente, ocasionando menos rotatividade e fluxo de desenvolvimento.

Desvantagens da coreografia sobre a orquestração

Complexidade

Na orquestração, assim como nos sistemas monolíticos, a lógica de negócios está centralizada, o que é mais fácil de manter e gerenciar.

Confiabilidade

As plataformas de orquestração possuem suporte integrado para tratamento de erros e gerenciamento de transações de compensação. Na coreografia, o fluxo de trabalho desenvolvido de forma personalizada e o tratamento de erros tendem a ser mais sujeitos a erros.

Rastreamento

O fluxo de negócio está explicitamente modelado e centralizado no orquestrador, por consequência o monitoramento de ponta a ponta e o rastreamento de mensagens são mais fáceis.

Discordância

Diante da ausência de um tomador de decisões, os integrantes do fluxo têm o direito de tomar decisões por si só. Esse direito pode levar a contradições quando um serviço atropela a decisão de outro, ou pior, os problemas com computação concorrente podem dar as caras.

Sim, estou falando das condições de corrida, *deadlocks* e *livelocks*. Nos Vingadores, vimos isso acontecer várias vezes, como na Guerra Civil, mas aqui, isso não é facilmente resolvido com um estalar de dedos.

19.4 Abordagem híbrida

Enquanto a orquestração usa uma abordagem centralizada para executar as decisões e ter melhor controle, a coreografia dá mais liberdade para executar essas decisões.

No meio termo, a orquestração e coreografia não precisam ser mutuamente exclusivas. Há uma terceira via, uma opção escondida na moita, um mix entre as duas formas de coordenação: a ~~**orquestreografia**~~ **abordagem híbrida**. Com ela, podemos usar ambos os modos de coordenação para obter outros resultados.

Coreografia com orquestração interna

Na ilustração a seguir, os serviços `A`, `B`, `F` e `G` são coreografados entre si e não percebem a existência de um orquestrador disfarçado entre eles. O serviço `Orquestrador` consome a mensagem *3*, que o aciona para orquestrar as chamadas *4.1*, *4.2* e *4.3* para os serviços `C`, `D` e `E`, respectivamente.

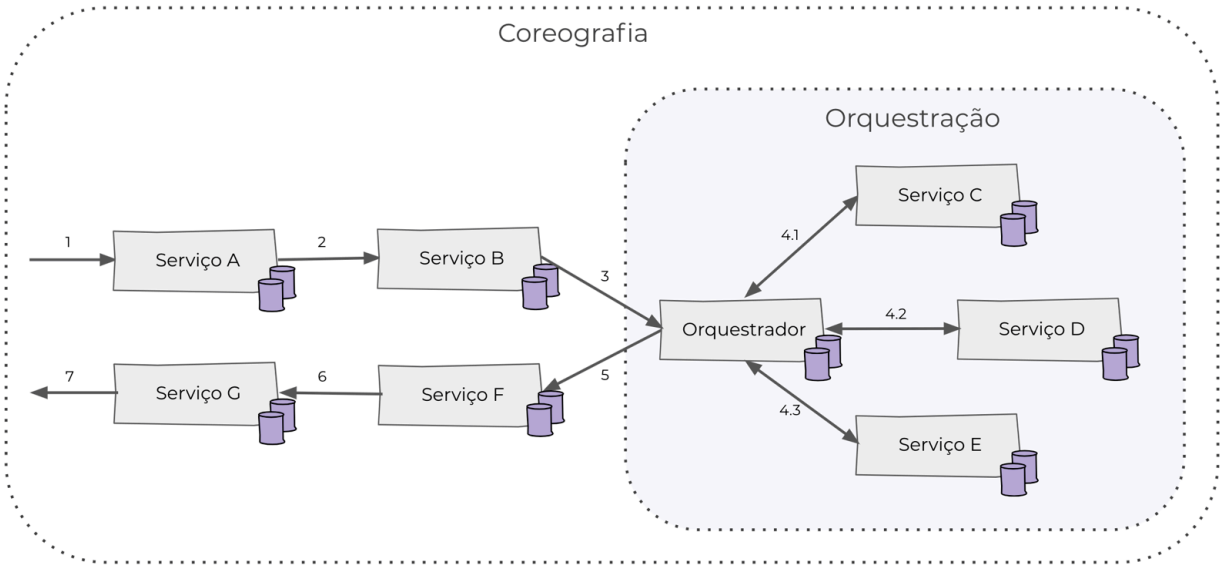

Figura 19.7: Coreografia com orquestração.

Por se tratar de orquestração, essas mensagens exigem respostas que podem ser assíncronas ou síncronas. O serviço `Orquestrador`, por meio da mensagem *5*, informa ao serviço `F` o resultado de seu processamento.

Orquestração com coreografia interna

Outra maneira de combinar os dois modos com uma abordagem híbrida é a oposta. Utilizar uma orquestração com uma coreografia interna. O segundo padrão híbrido usa um orquestrador para gerenciar o fluxo e uma coreografia no papel de um serviço qualquer.

O coordenador produz o comando *5.1* para ordenar uma ação do serviço `D` e recebe um evento do serviço `G` indicando o resultado da operação coreografada, como mostra a ilustração:

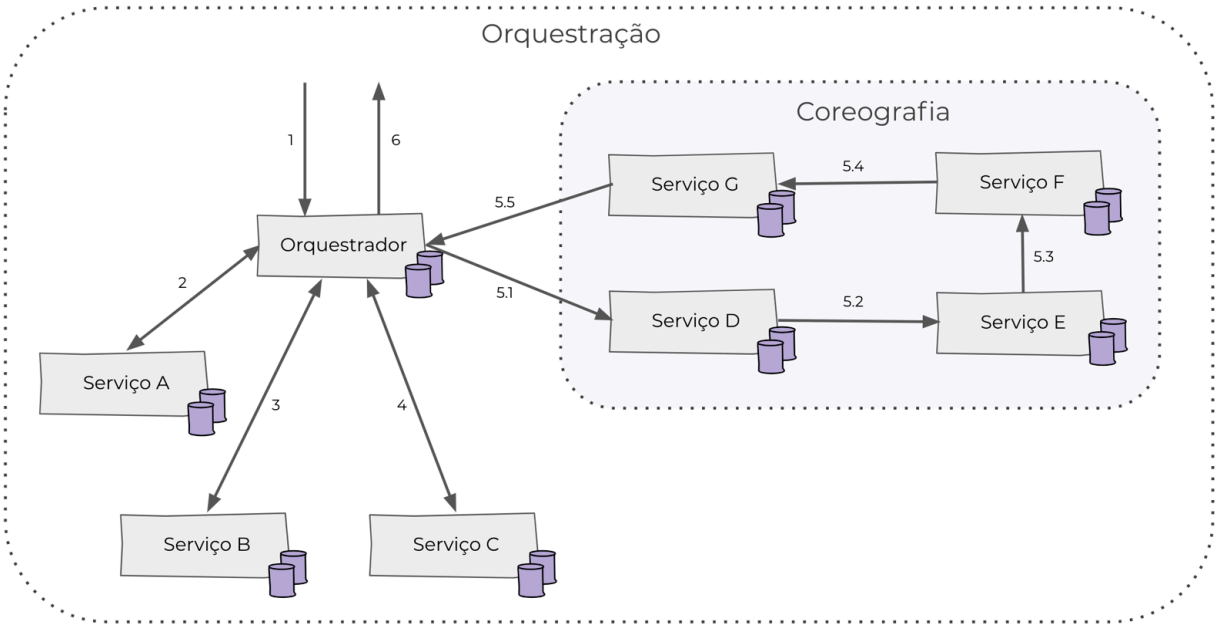

Figura 19.8: Orquestração com coreografia.

A abordagem híbrida permite a união do melhor dos dois mundos. Em um modelo de negócios complexo por natureza, podemos aplicar um orquestrador para facilitar seu gerenciamento e controle. No conjunto coreografado de serviços, a coordenação é escalável e não há SPOF.

Em contrapartida, a abordagem também une o que há de pior nos modos de coordenação. O fluxo geral é distribuído entre todos os orquestradores e as coreografias envolvidas. Por possuir um orquestrador, a abordagem é necessariamente mais acoplada do que uma inteiramente coreografada.

O desempenho e a escalabilidade serão limitados aos orquestradores e eles continuam sendo pontos vulneráveis na arquitetura. Apesar de não serem mais SPOF, eles ainda são gargalos com maior valor de negócio do que os meros mortais minions.

Coordenação com mediadores

Um ponto a ser notado é a ausência de relação entre a forma de coordenação do fluxo com protocolos de comunicação e *brokers*. As

abordagens podem ser implementadas de maneira síncrona e assíncrona e com ou sem *brokers*/mediadores.

Uma ressalva é que, caso haja uma ferramenta mediadora: 1) ela não deve atuar como **ESB** (*Enterprise Service Bus*, ou Barramento de Serviços Corporativos), um antipadrão observado na indústria que centraliza esquemas e lógica de negócios; e 2) ela deve respeitar a filosofia de **smart endpoints, dumb pipes** (pontos de extremidade inteligentes e tubos tolos, do inglês), lema que incentiva um design mais desacoplado com cada microsserviço tendo seus próprios dados e sua própria lógica do domínio.

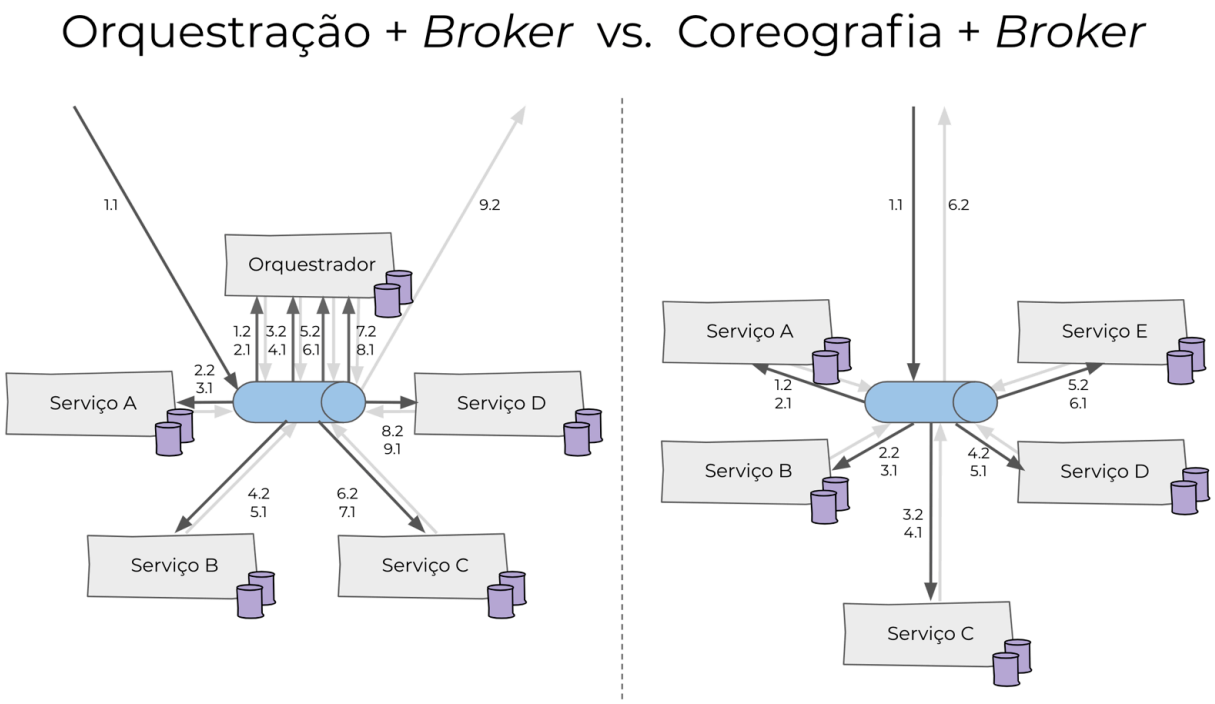

Figura 19.9: Orquestração com mediador vs. Coreografia com mediador.

19.5 Conclusão

O orquestrador gerencia centralmente a resiliência do fluxo de trabalho e pode se tornar um único ponto de falha. Na coreografia, a

função é distribuída entre todos os serviços, e o controle da operação é pulverizado.

Isso significa que uma coreografia difere de uma orquestração no que diz respeito a onde deve residir a lógica que controla as interações entre os serviços envolvidos.

No geral, todas as três categorias de coordenação (orquestração, coreografia e híbrida) podem ser aplicadas em diferentes cenários e casos de uso.

Por preferência pessoal e por se tratar de uma **perspectiva reativa**, costumo dar preferência ao uso da coreografia e inserir orquestrações quando necessário. Entendo, todavia, que essa se trata da minha opinião e reconheço que eu tenho uma queda por esse tema. Sei lá, o jeito como os sistemas reativos me olham... É diferente.

A abordagem adotada refletirá em como o erro será tratado. Uma maneira é fazer com que o serviço indique falha disparando um evento. Outro serviço, possivelmente o orquestrador, recebe o evento e executa as ações necessárias, como aplicar transações de compensação em cadeia para desfazer operações interrompidas. Outra solução é **tentar novamente** a ação anterior, ou usar mecanismos de **tempo limite** para aceitar a falha.

Indiferentemente do modo de coordenação escolhido, é importante haver persistência da máquina de estados. O local da persistência varia de acordo com o modo selecionado. Havendo persistência, se um serviço não conseguir concluir uma operação, será possível recuperar e dar continuidade à operação.

Capítulo 20
Observabilidade de sistemas

Figura 20.1: Observabilidade.

Os sistemas resilientes enfrentam o desafio de suprir a demanda no ritmo exigido. Um ambiente complexo, como o dos microsserviços, é muito convidativo para monstruosos erros, frutos de combinações inimagináveis de fatores que podem dar errado.

No mundo real, a verdade é desanimadora – como ela sempre costuma ser em qualquer outro lugar. Na prática, os microsserviços se tornam coisas bem diferentes dos slides ou dos livros que falam a respeito disso. Viram um caos.

Temos aplicações de tudo quanto é tipo, conectadas a outras aplicações de toda natureza, o que torna a tarefa de executar qualquer mudança, seja uma implementação de melhoria,

atualização de segurança ou adição de um novo sistema, uma orgia de riscos entre os cavaleiros do apocalipse.

Sistemas reativos, com sua independência de deploy e indecência de mensagens assíncronas transitando para tudo que é lado, fazem com que esse ambiente seja ainda mais confuso e tornam a utilização de algum método de controle ainda mais importante.

Nesses ambientes caóticos, a simples ideia de adição de features é encarada como se um urso panda fosse decapitado durante uma live do Greenpeace.

A preocupação é óbvia, o negócio pode ser impactado, assim como seriam os lucros e a reputação que esses resultados produzem. Uma falha arruína o valor da marca e demanda tempo e dinheiro para um conserto. O tempo de atividade, a escalabilidade e a capacidade de identificar e corrigir erros são os fatores que determinam se você permanecerá no negócio ou se perderá para a concorrência.

Os riscos não podem ser eliminados, mas podem ser minimizados. Felizmente, vivemos na era dos aplicativos observáveis em execução em infraestrutura observável.

Entender como o sistema está se comportando em produção não é um exercício abstrato. É possível ter um modelo preciso e baseado em fatos do comportamento da aplicação para nortear a tomada de decisão humana a fim de garantir que a empresa esteja pronta quando o pior acontecer.

Além disso, quando ocorre um desastre, as equipes são forçadas a vasculhar seus dados para determinar o que aconteceu e garantir que não aconteça novamente.

As atividades usuais de auditoria podem envolver incontáveis dias de desenterramento de dados de fontes distintas. Uma plataforma ideal teria um sistema de "caixa-preta" de registro de cada ação realizada. O quê, quando e o porquê precisam ser respondidos para evitar que uma nova catástrofe ocorra no futuro.

Somente um bom entendimento do comportamento do sistema em produção fornecerá a segurança e a tranquilidade que os desenvolvedores precisam para acompanhar o ritmo necessário.

À revelia das suas melhores intenções, implantar uma nova versão, iniciar uma migração de dados, ou até mesmo corrigir um bug sem um bom acompanhamento dos serviços é tão seguro quanto saltar de um avião usando uma geladeira como paraquedas. Não é que não funcione, mas só funciona quando o avião não está voando, e olhe lá.

Figura 20.2: Sistemas reativos, não confundir com sistemas refrigerativos.

20.1 Observabilidade

O que as equipes precisam é de uma imagem em tempo real de cada aspecto do processo de mudança à medida que ele se desenvolve, permitindo identificar e eliminar mais rapidamente os potenciais riscos. Essa imagem também deve fornecer percepções sobre a utilização de recursos, desempenho de metas e desperdício de tempo e precisão.

Atingir esse nível de observabilidade e ter visibilidade de todos os eventos são fundamentais para o sucesso de um projeto.

Observabilidade não é uma ferramenta e não é telemetria. Observabilidade é mais que um painel reluzente que retrata visualmente o estado de seus aplicativos e infraestrutura. É a paz de espírito que os mantenedores do sistema merecem. Para isso, você precisa de mecanismos que informem que o sistema está no ar, e quais são as informações relevantes para sua operação.

Resiliência e observabilidade são dois pontos de vista diferentes do mesmo problema e, neste capítulo, farei um tour pelos conceitos a serem considerados para a construção de uma solução de observabilidade.

Monitoria vs. Observabilidade

Segundo Cindy Sridharan, que irradia cognição no seu sucinto livro *Distributed Systems Observability*, não dá para falar sobre observabilidade sem diferenciá-la de monitoria.

Seus objetivos são diferentes. Monitoria fornece dados das aplicações em execução. Números, taxas, requisições por minuto, utilização de CPU e gráficos cintilantes são monitoria. Ou, melhor dizendo, são sinais de monitoria. A observabilidade engloba a monitoria, por isso ela não substitui e nem elimina a necessidade da monitoria.

A observabilidade fornece não apenas visões gerais de alto nível da integridade do sistema, mas também percepções granulares sobre os modos de falha dele. Além disso, um sistema observável fornece

amplo contexto sobre seu funcionamento interno, desbloqueando a capacidade de descobrir questões sistêmicas mais profundas e entender qualquer problema que surja.

Monitoria mínima

Existe uma saraivada de siglas e métodos que instruem qual deve ser o conjunto mínimo de métricas usadas para fins de monitoria:

- **Quatro sinais de ouro:** os autores do livro *Engenharia de Confiabilidade do Google* dizem que, em cada serviço, os sinais mínimos viáveis para uma boa monitoria para fins de alerta são a **latência**, os **erros**, o **tráfego** e a **saturação**.
- **Metodologia USE:** Brendan Gregg (em: http://www.brendangregg.com/usemethod.html) diz que, para cada recurso, o mínimo a ser monitorado é o **uso**, porcentagem de tempo que o recurso permaneceu ocupado, memória disponível, nível de utilização da CPU; **saturação**, quantidade de trabalho que o serviço tem que fazer, normalmente o tamanho da fila; e, por último, os **erros**, a quantidade de eventos de erros.
- **Metodologia RED:** proposto por Tom Wilkie (veja em: https://www.weave.works/blog/the-red-method-key-metrics-for-microservices-architecture/), o método exige, para uma monitoria mínima de um recurso, as métricas de: **taxas** (no inglês, *rate*), número de requisições por segundo; **erros**, a quantidade de requisições que falham; e **durações**, a quantidade de tempo que essas requisições duram.

Use a que lhe for mais conveniente, ou nenhuma delas, ou uma permutação entre elas. Não se sinta pressionado. O que importa é que, no fim das contas, você possua bons números para relatar a integridade geral dos sistemas e gerar alertas de qualidade.

20.2 Alertas

Os alertas não são apenas para incidentes de infraestrutura, como falta de recursos e falhas, eles também devem ser usados para notificar os proprietários da plataforma sobre eventos excepcionais que o sistema não pode manipular sozinho.

Um alerta costuma ser um recurso de uma ferramenta de monitoria para informar a um humano sobre uma situação que precisa de atenção.

É fundamental ter o sistema de alerta integrado ao seu sistema de resposta a incidentes e ao serviço de geração de tarefas. Isso gera itens de ação automaticamente e garante, em primeira instância, que o alerta não cairá no esquecimento. Na pior das hipóteses, a tarefa tem que ser deliberadamente jogada para escanteio.

Figura 20.3: Estado de alerta depois de uns tapas na pantera.

Considere um *e-commerce* que recebeu o pagamento de um produto, mas teve uma falha interna antes que o pedido fosse

totalmente processado. Configurar um alerta que registra um incidente com o sistema de gerenciamento de tarefas para que uma equipe humana resolva a falha pode ajudar a resolver o problema de forma proativa, sem que o cliente final fique revoltado, tome medidas agressivas e ameace processar a empresa.

O alerta é intrinsecamente centrado no fracasso e no ser humano. Os sistemas mais distribuídos levaram à criação de ferramentas e plataformas sofisticadas para a abstração vários dos problemas que a monitoria humana centrada em falhas ajudou a descobrir.

Verificação de integridade, balanceamento de carga e disparo de disjuntores, como falamos no capítulo *Fluxos resilientes*, são recursos que algumas plataformas fornecem para uso imediato, liberando os operadores da necessidade de ser alertados.

Assim como os testes no geral, alertas devem ser levados a sério. Dessa forma, ao menor sinal de que um alerta está sendo ignorado, sua necessidade ou sua configuração deve ser revista.

É como o alarme de segurança do condomínio onde eu moro. Ele alarma cinco vezes por dia, no mínimo. Em algum momento, um ladrão de verdade vai entrar no prédio e só vão perceber horas depois. Nesse caso, não ter alarme seria até melhor pois não criaria uma falsa sensação de segurança. Agora me respondam, qual é a utilidade de alarmes assim?

20.3 Pilares da observabilidade

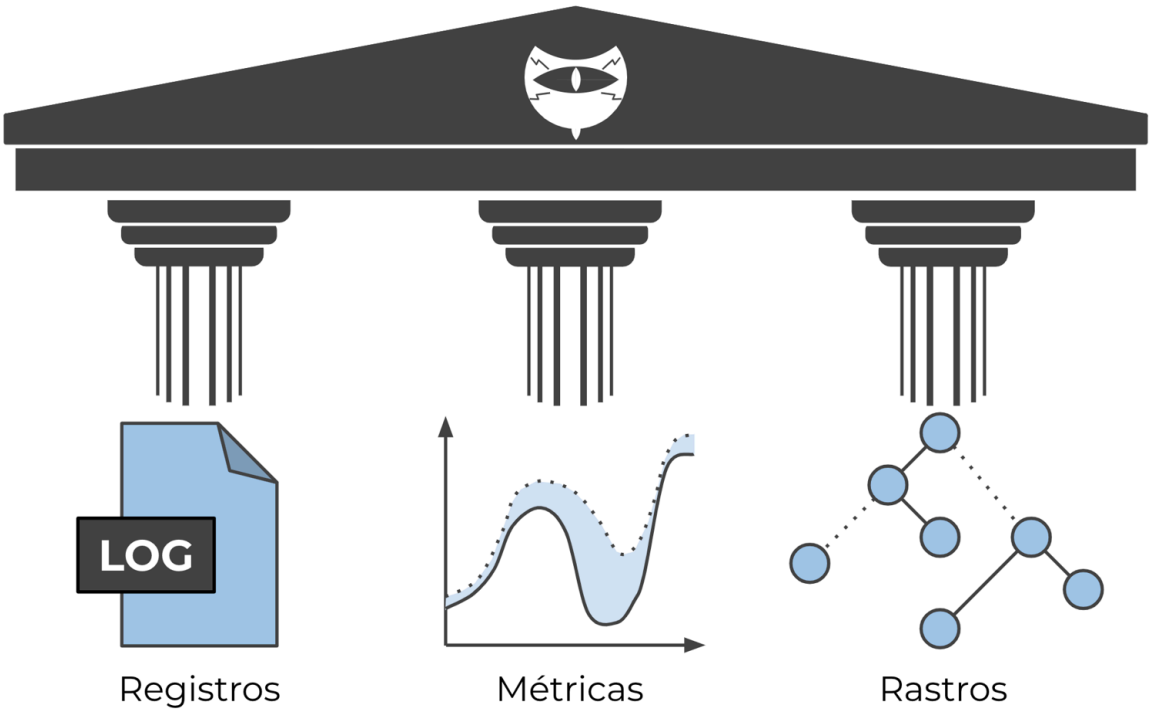

Figura 20.4: Três pilares da observabilidade.

Ainda citando Cindy Sridharan e seu conciso livro, a observabilidade possui três pilares, sendo eles: **registros de eventos**, **métricas** e **rastreamento**.

Os pilares são ferramentas poderosas que, se bem compreendidas, habilitam a construção de sistemas melhores. Mas ter os três pilares de pé não torna o sistema mais observável. É importante que eles sejam robustos e sejam capazes de suportar toda a estrutura da observabilidade e, principalmente, que sejam capazes de extrair respostas do sistema.

Transformar dados em respostas requer mais do que apenas um sistema observável.

Registros de eventos (ou *event logs*)

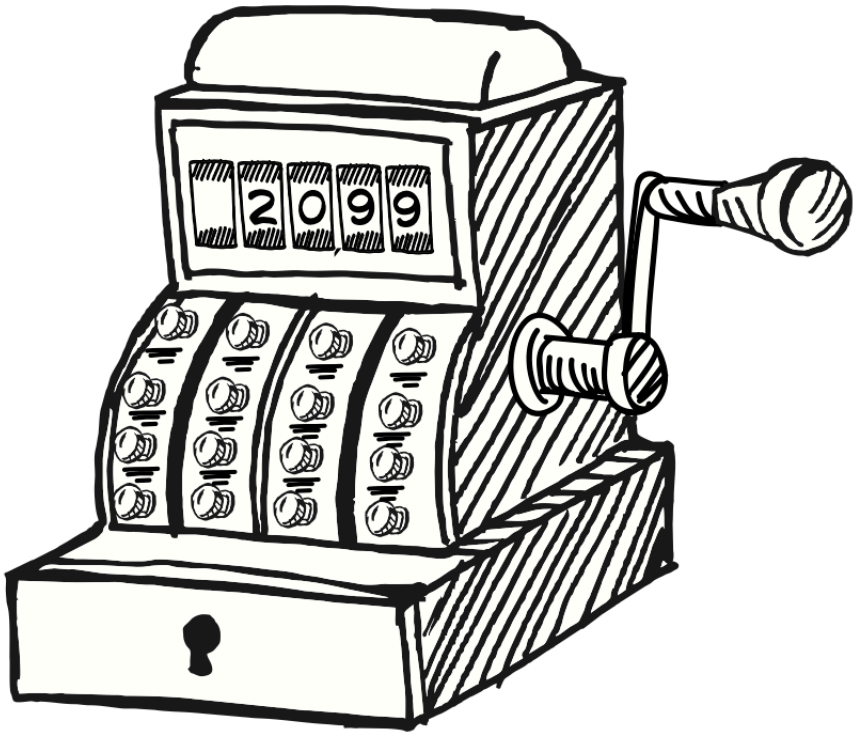

Figura 20.5: Registro de eventos à moda antiga.

O primeiro pilar é o *event log*, um registro imutável com carimbo de data e hora de algum contexto sobre o que aconteceu ao longo do tempo. Os logs de eventos em geral vêm em três formas, todas úteis:

1. **Texto simples:** um texto de formato livre. Este é o formato mais comum de registros.
2. **Estruturada**: forma muita evangelizada e defendida nos últimos anos. Normalmente, esses logs são emitidos no formato JSON ou similar e possuem a vantagem de permitir buscas rebuscadas.
3. **Binário**: formas estruturadas serializadas. Esta forma implica em logs menores, logo menos espaço de armazenamento gasto, e logs transmitidos de maneira mais rápida e não legíveis por humanos, pelo menos não com os quais eu convivo.

A depuração, dependendo da raridade do problema, requer um nível mais alto de granularidade. Os registros de eventos, em particular, brilham quando se trata de fornecer uma visão valiosa e um amplo contexto sobre a cauda longa que médias e percentis não apresentam.

Como dito no capítulo *Inconsistência distribuída*, os logs de eventos são especialmente úteis para descobrir comportamentos emergentes e imprevisíveis exibidos por componentes de um sistema distribuído.

Os logs são, de longe, o pilar mais fácil de manusear. O fato de o log ser um dado de fácil representação e de ser de fácil instrumentação, já que adicionar uma linha de log é trivial, colabora bastante para isso.

Muitas linguagens de programação, bibliotecas e frameworks possuem implementação bem estabelecida e em conformidade com os padrões do mercado. A utilização exacerbada de logs, no entanto, pode afetar adversamente o desempenho da aplicação. Embora a geração de log seja fácil, algumas bibliotecas enfraquecem o desempenho devido à sobrecarga e ao processamento não assíncrono e bloqueante de sua gravação.

Fora o desempenho, armazenar uma quantidade massiva de registros pode ser tão caro quanto um rim bem conservado no mercado paralelo – não me perguntem como eu sei essa informação. Por esse motivo, é bastante comum optar por uma retenção desses dados por poucas semanas ou até mesmo dias.

Por sua facilidade de implementação, é comum que dados de registros de evento não sejam usados exclusivamente para os casos de uso de depuração, uma vez que o processamento de log também se encaixa perfeitamente na conta de **Processamento Analítico On-line** (no inglês: *Online Analytical Processing*, ou apenas OLAP). Por isso, eles também podem ser fonte de todos os dados analíticos, junto com as métricas.

Métricas

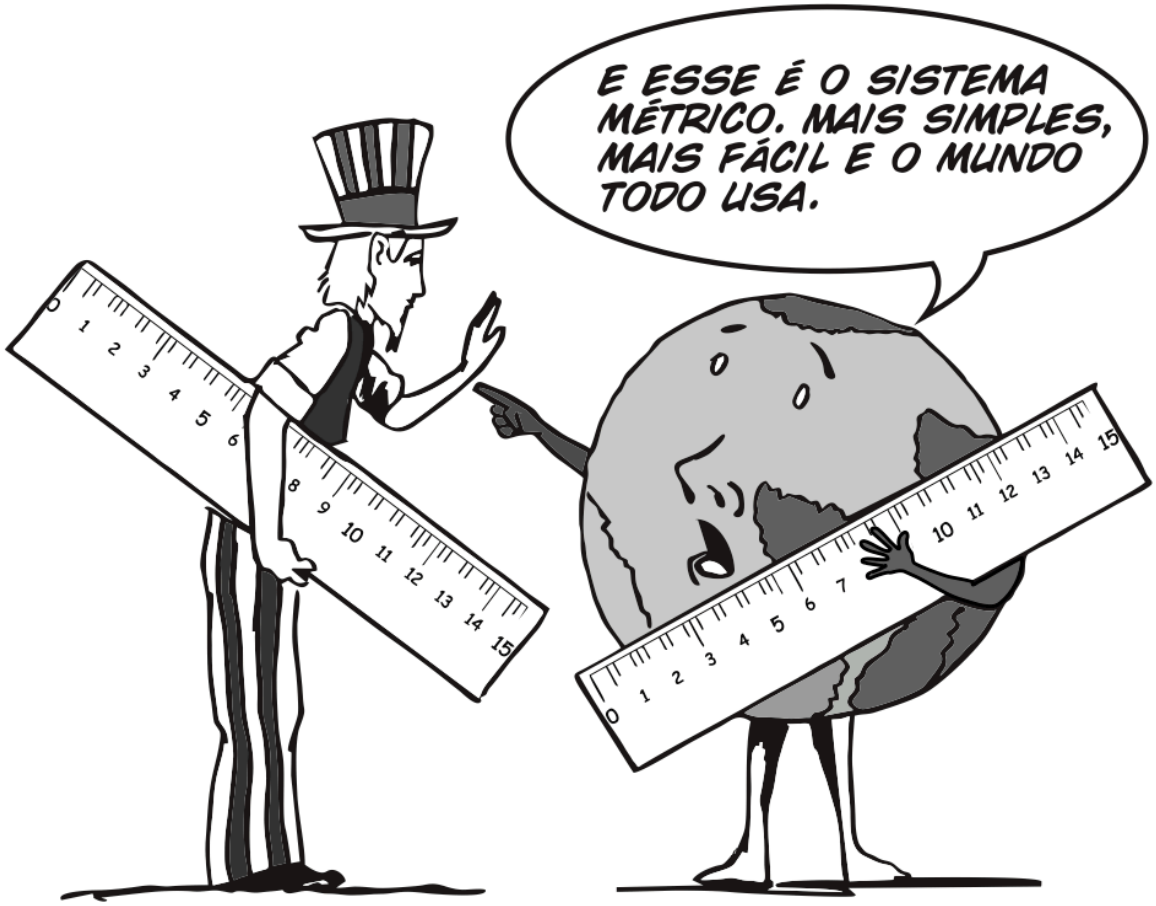

Figura 20.6: Sistema imperial vs. sistema métrico?

As métricas são o segundo pilar da observabilidade. Elas são uma representação numérica dos dados medidos em intervalos de tempo.

Métricas podem aproveitar o poder da modelagem matemática para derivar o conhecimento do comportamento de um sistema em intervalos de tempo no presente e no futuro. São muito úteis como técnica para alcançar uma boa escalabilidade dinâmica, como abordado no capítulo *Jogo de dados*.

Como os números são otimizados para armazenamento, processamento, compactação e recuperação, as métricas permitem uma retenção mais longa de dados bem como consultas mais fáceis.

Isso as torna mais adequadas para a construção de painéis que refletem tendências históricas, do que registros de eventos.

Uma vantagem das métricas sobre o registro de eventos é que, ao contrário da geração e do armazenamento de logs, a transferência e o armazenamento de métricas pode ter uma sobrecarga constante.

Ao agregar as métricas do lado do cliente, podemos garantir que o tráfego não aumente proporcionalmente ao tráfego do usuário.

Desse modo, o custo das métricas não aumenta em função do tráfego do usuário ou qualquer outra atividade que resulte em uma ampliação acentuada nos dados. Alguns exemplos de quando o custo pode aumentar são: a quantidade de valores de rótulos, quantidade de servidores ou contêineres ativados, adição de novos serviços ou acréscimo de mais métricas nos serviços existentes.

Uma desvantagem das métricas em relação aos registros de eventos é que as métricas possuem uma forte tendência para transformações matemáticas, probabilísticas e estatísticas. Não que isso seja ruim, mas, por conta disso, manusear ferramentas de visualização de métricas pode se tornar uma tarefa árdua e exigir conhecimentos não triviais. Fora ter, no mínimo, dois Nobels em matemática, um curso dos bons no SEBRAE, umas três copas do mundo e as seis joias do infinito.

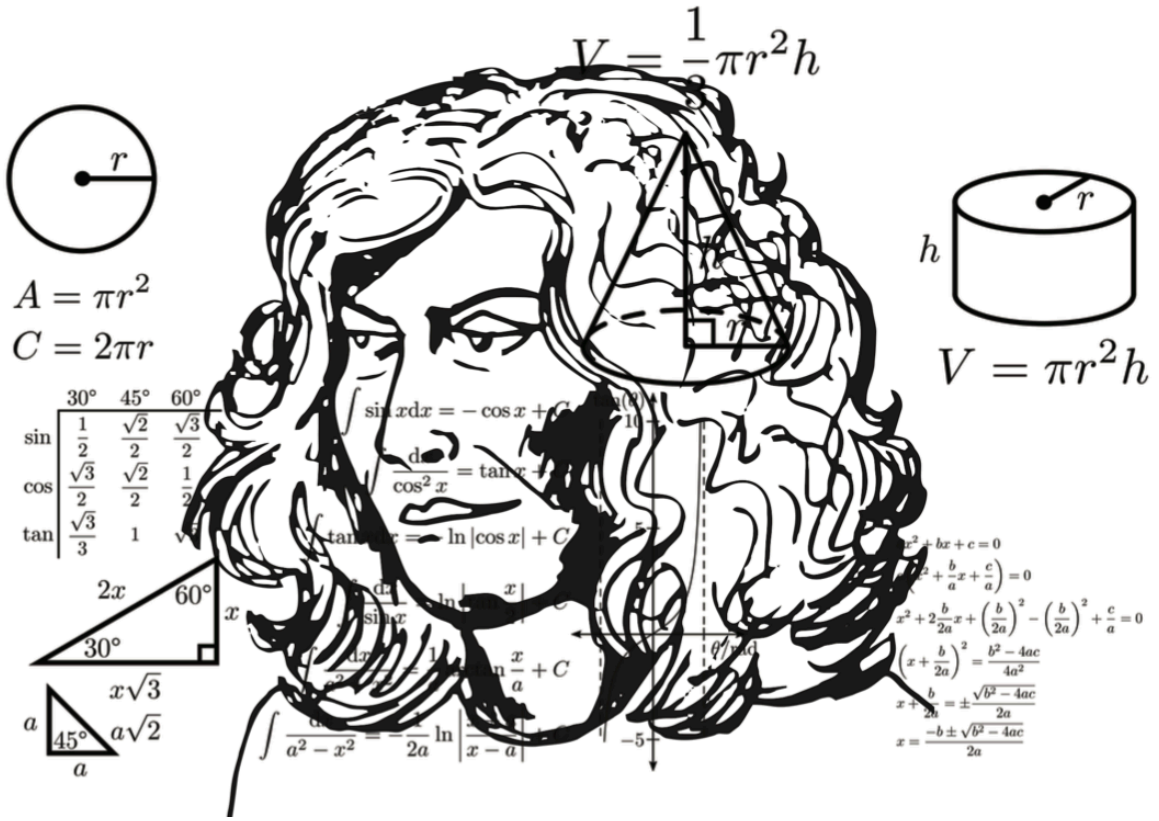

Figura 20.7: Desenvolvedor criando dashboards de métricas.

Uma vantagem é que essas características matemáticas tornam as métricas mais adequadas para relatar a integridade geral de um sistema e para acionar alertas, visto que executar consultas em um banco de dados na memória é muito mais eficiente e confiável do que executar uma consulta em um sistema distribuído e agregar os resultados para decidir o acionamento de um alerta.

Quando usados de maneira ideal, os registros e as métricas nos fornecem a onisciência de um escopo.

Embora bastem para compreender o desempenho e o comportamento de sistemas individuais, os dois primeiros pilares não são suficientes para entender o tempo de vida e o comportamento de um fluxo que envolve vários sistemas. Com essa finalidade, existe o terceiro pilar: o rastreamento distribuído.

Rastreamentos (ou *traces*)

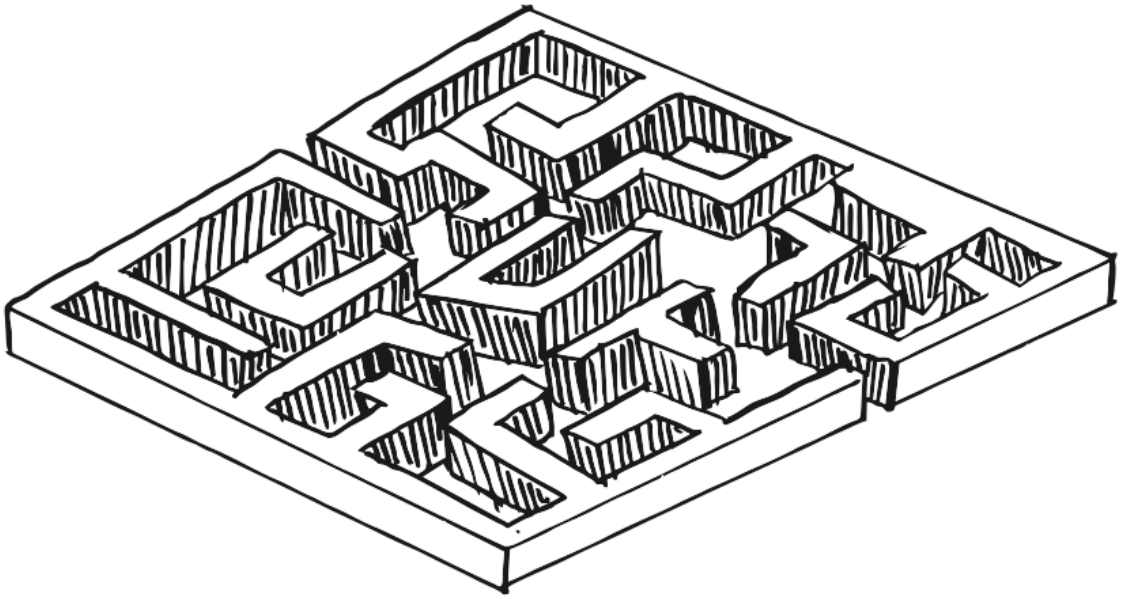

Figura 20.8: Rastreamento distribuído.

O rastreamento é uma representação, geralmente visual, de uma série de eventos distribuídos e causalmente relacionados, que mostra um fluxo de ponta a ponta em um sistema distribuído.

A ideia por trás do rastreamento é simples: identificar pontos específicos em um fluxo, serviço, biblioteca, middleware e qualquer outra coisa que possa representar bifurcações no fluxo de execução.

Os rastreamentos são usados para identificar a quantidade de trabalho realizado em cada camada, preservando a causalidade entre os eventos. Um dos resultados é um grafo acíclico e direcionado, em que cada vértice é um componente participante, um ponto de instrumentação, e as arestas são as chamadas entre eles.

Ter uma compreensão de todo o ciclo de vida da solicitação torna possível depurar solicitações que abrangem vários serviços para identificar a origem do aumento da latência ou utilização de recursos.

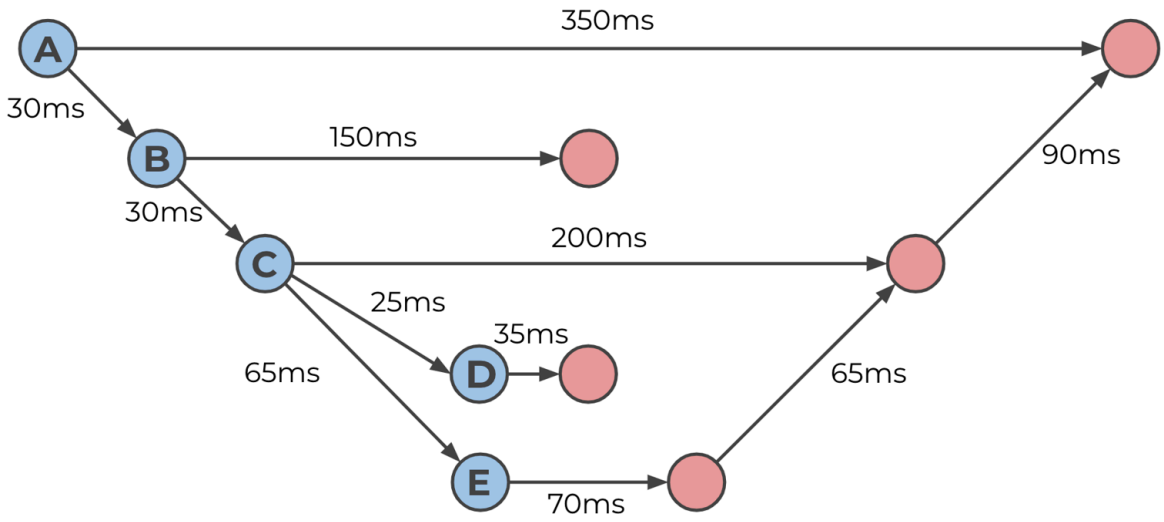

Figura 20.9: Exemplo do grafo mencionado.

Quando uma solicitação começa, é atribuída a ela uma série de identificadores como de sessão e de transação e um identificador de correlação globalmente exclusivo (assunto mencionado também nos capítulos *Inconsistência distribuída* e *Mensagens distribuídas*), que é propagado por todo o fluxo para que cada ponto de instrumentação seja capaz de enriquecer os metadados de rastreamento antes de repassar o identificador para o próximo salto.

Esses dados são suficientes para a reconstrução do fluxo de execução com base em diferentes valores emitidos por várias partes do sistema. Por ele, percebemos quais componentes interagem e estão envolvidos no fluxo, qual ordem, a duração de cada etapa e outras coisas.

Coletar essas informações e reconstruir o fluxo de execução para análise retrospectiva e solução de problemas permite que se compreenda melhor o ciclo de vida de uma solicitação.

O custo do rastreamento não é tão agressivo quanto o da utilização massiva de registro de eventos, principalmente porque os rastreamentos podem ser feitos por amostragem e reduzem a sobrecarga do tempo de execução, bem como os custos de

armazenamento. Isso é uma excelente notícia se você quer preservar seu rim dentro do seu corpo.

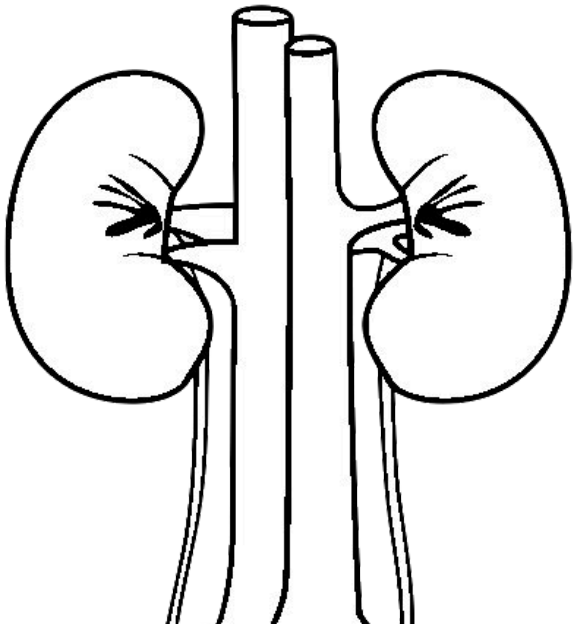

Figura 20.10: Usar menos logs poupa rins.

Nos sistemas reativos, o rastreamento tem um valor ainda maior do que o habitual. A Orientação a Mensagens assíncronas e o ziguezaguear das mensagens sendo atiradas a todo momento em toda direção faz com que a tarefa de mapear esses fluxos manualmente beire o impossível e o produto do mapeamento fique rapidamente incorreto, obsoleto e inútil.

O problema do rastreamento reside na dificuldade de implementação. O rastreamento é, de longe, o mais difícil de adaptar em uma infraestrutura existente.

Para que o rastreamento seja realmente eficaz, cada componente do fluxo precisa ser modificado para propagar as informações. Para um bom resultado, não é suficiente que os desenvolvedores instrumentem seu código; os componentes utilizados pelos sistemas e aqueles que o utilizam também devem ser instrumentados.

Isso é bem desafiador em arquiteturas poliglotas e em diversas, como a de microsserviços, com vários SGBDs diferentes, linguagens de programação e protocolos de comunicação. Felizmente, quando os componentes envolvidos se tratam de recursos praticamente onipresentes no mercado, algumas ferramentas costumam dar conta do trabalho. Nesse cenário, ferramentas como Zipkin e Jaeger em conjunto com o Spring Cloud Sleuth requerem quase nada de configuração e "automagicamente" produzem relatórios assustadoramente completos.

Figura 20.11: Isso é mágica.

20.4 Conclusão

Monitoria e observabilidade não são a mesma coisa. Observabilidade vai além, é o olho que tudo vê. O estado de consciência plena de tudo que envolve a arquitetura. É a capacidade de antecipar e de se

preparar para uma situação de risco, é quase como um sexto sentido. Como diria Ross Geller em *Friends*, é ter *unagi*.

Figura 20.12: Rolinho de pele de salmão.

Observabilidade é a trilha de migalhas de pão que o próprio sistema deixa para o alcance da resiliência. Não é possível tornar o sistema resiliente se você não sabe o que o derruba.

Como diria meu professor de Análise e Projeto de Software, cujo nome e o restante da disciplina nunca mais me voltarão à memória: "não dá para melhorar o que não se mede". Vimos que, sem uma observabilidade de primeira, desenvolver um sistema é como um voo cego.

A observabilidade é subdividida em três pilares: registros de eventos, métricas e rastreamento. Cada um atende a uma única finalidade e são complementares entre si. Em uníssono, eles fornecem visibilidade máxima sobre o comportamento de sistemas distribuídos. Somente assim, os desenvolvedores poderão ter o

sossego que precisam para inserir de forma não proposital os bugs mais difíceis de serem encontrados.

A observabilidade não se resume apenas a seus três pilares. Simplesmente tê-los não resulta em sistemas observáveis. A observabilidade é um esforço constante e, ainda assim, é necessário se conformar que, por mais observável que seja, o sistema nunca estará totalmente saudável e nunca será completamente previsível.

São muitas as ferramentas disponíveis no mercado. É comum que a mesma ferramenta contemple mais de um pilar da observabilidade. Correndo um grande risco de defasar rapidamente a lista a seguir, cito ferramentas por meio das quais tive boas experiências e que podem ser pesquisadas se você quiser aprofundar o seu conhecimento:

- New Relic (https://newrelic.com/)
- Dynatrace (https://www.dynatrace.com/)
- Splunk (https://www.splunk.com/)
- Elastic Stack (https://www.elastic.co/)
- Prometheus (https://prometheus.io/)
- Grafana (https://grafana.com/)
- Datadog (https://www.datadoghq.com/)
- Fluentd (https://www.fluentd.org/)
- e as já mencionadas Zipkin e Jaeger (https://zipkin.io/ e https://www.jaegertracing.io/).

Posso assegurar com serenidade que me esqueci de várias e, conforme você for pesquisando, você vai perceber que outras dezenas poderiam ser adicionadas à lista. No entanto, eu teria que parar em algum momento, não é?

Uma combinação de diferentes sinais de observabilidade vinculados semanticamente permitem um bom nível de depuração, conforto aos desenvolvedores e, principalmente, embasamento para as decisões de negócio.

Explicarei o real valor do embasamento para o negócio no próximo capítulo.

Parte 7: Responsividade

Capítulo 21
O negócio e a responsividade

A responsividade é um conceito da Ciência da Computação que se refere à habilidade de um sistema de completar suas tarefas dentro de um determinado tempo.

Sistemas que são responsivos se concentram em fornecer tempos de resposta rápidos e consistentes, estabelecendo limites superiores para fornecer uma qualidade confiável. Esse comportamento simplifica o tratamento de erros, aumenta a confiança do usuário final e incentiva mais interação.

Apesar da relação, a responsividade não é exclusivamente sobre desempenho. Um hardware lento é capaz de executar um software responsivo assim como um software rápido, mesmo se executado pelos computadores *master blaster deluxe* do Pentágono, pode ser tão responsivo quanto uma tartaruga marinha dopada.

Figura 21.1: A gente nunca saca, irmão. Mas quando eles sacam, aí tu saca, sacou?

A tolerância do usuário e o nível de frustração variam dependendo da situação. É relativamente aceitável que uma transação financeira dure mais do que alguns segundos.

Por outro lado, é inaceitável que adicionar um item ao carrinho de compras de um *e-commerce* requeira o mesmo tempo. Pior, a falta de responsividade, nesse caso, pode levar à perda da venda e, em último caso, do usuário. Para um *e-commerce*, isso pode ser fatal e determinar sua própria sobrevivência.

Longos atrasos podem ser uma das principais causas de frustração do usuário. Podem levá-lo a acreditar que o sistema não está funcionando ou que um comando foi ignorado.

Por isso, a **capacidade de resposta** é considerada uma questão essencial de usabilidade. A lógica é que o sistema deve entregar os resultados de uma operação aos usuários de maneira oportuna e organizada.

21.1 Capacidade de resposta

A capacidade de resposta tem na sua base, implicitamente, a **disponibilidade** e a **elasticidade**. Ela tem como objetivo melhorar a usabilidade do usuário. Para ter uma boa capacidade de resposta, é importante que um sistema gaste os recursos disponíveis da melhor maneira possível.

Por exemplo, faz sentido permitir que um mecanismo como o mouse possua alta prioridade para fornecer interações fluidas com o usuário. Em operações de longo prazo, como transações financeiras, baixar ou transformar arquivos grandes, o fator mais importante é fornecer um bom feedback e não o desempenho da operação, já que pode muito bem ser executado em segundo plano.

Na teoria, soa simples. O desafio implica em lidar com sistemas reais. Coisas que precisam ser mantidas, atualizadas, implantadas, revisadas etc. Não se trata de prova de conceitos experimentais de um final de semana ou trabalhos de faculdade.

No mundo real, o bicho pega e as aplicações têm chance de falhar. A implacável lei de Murphy diz que tudo que tem a menor chance de falhar certamente falhará. Gosto de complementar dizendo que falhará no pior momento possível e da pior maneira possível.

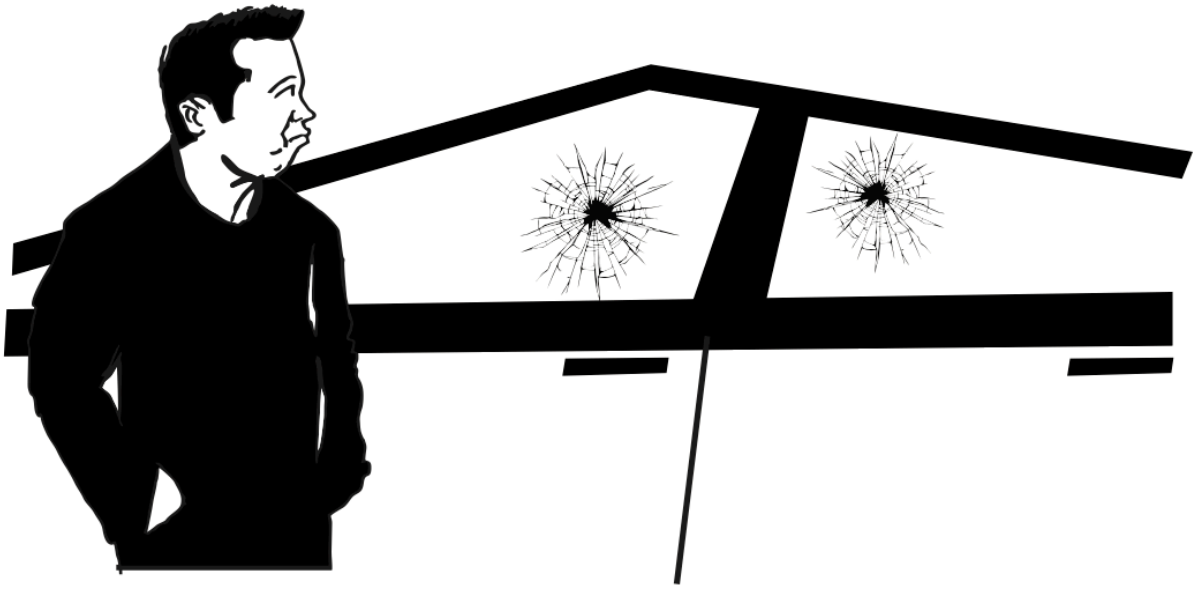

Figura 21.2: Elon Musk sabe do que eu estou falando.

Existe uma espécie de mito sobre a capacidade de resposta que nos diz que um dos primeiros passos para ter serviços responsivos é escolher uma ótima tecnologia. A realidade nos diz, contudo, que essas tecnologias são frequentemente escolhidas por tendência ou comodidade.

Não é a tecnologia que dita a responsividade. Sistemas não reativos podem muito bem ser mais responsivos do que sistemas reativos. Rótulos e títulos não ditam a responsividade se não implementados a sério.

O Instagram e o Pinterest provam que é perfeitamente aceitável escrever sites responsivos com tecnologias como Django, e a arquitetura do Stack Overflow dá um show de responsividade mesmo sendo monolítica.

21.2 Usuário vs. desenvolvedor

É aqui que habita um ponto crucial sobre o desenvolvimento de software. A comum falta de entendimento sobre o que é responsividade entre o usuário e o desenvolvedor.

Comumente, para quem desenvolve, a responsividade se trata de aguentar o tranco no horário de pico, da verificação do número de instâncias escalonadas, do suporte ao máximo de mensagens simultâneas e outras balelas técnicas. O desenvolvedor pensa sobre se a arquitetura atual será capaz de responder da mesma maneira daqui a alguns anos quando o volume de dados trafegado for muitas vezes maior.

A classe desenvolvedora de software tem a estranha compulsão de escrever o software mais perfeito, reutilizável e padronizado, que implemente todas as novas estruturas de ponta e esteja em conformidade com os padrões de arquitetura mais recentes.

O usuário não sabe nem dizer o que é responsividade. Nenhum usuário dirá: "Uau, este aplicativo consegue lidar com tantas requisições ao mesmo tempo!" ou "Você pode acreditar como este software é resiliente?". Por outro lado, se o sistema não for suficientemente responsivo, o chilique é certo, e coberto de razão.

O usuário não dá a mínima sobre a pilha tecnológica ou padrões adotados. Ele só quer ser feliz, curtir as fotos de sushi dos amigos, escrever "tá pago" e divulgar que o tio dele também faz caixa de sapato: "Interessados, entrar em contato por DM".

Querendo ou não, a única coisa pela qual o usuário se interessa é sobre a experiência ao interagir com o sistema.

21.3 O propósito do sistema

Nessa privação de consentimento, quem leva a melhor é o usuário, o negócio.

O que costuma ser relegado é que o propósito do desenvolvedor não é a tecnologia, é resolver um problema de negócio. Independentemente de o problema ser vender livros, vender benditas caixas de sapato, ou lançar naves tripuladas para Marte. A tecnologia é apenas o meio condutor, o veículo por meio do qual os objetivos de negócio serão atingidos.

É como diz um Guilherme Froes, com quem trabalhei por alguns anos, em https://guifroes.com/no-such-thing/. Seu objetivo enquanto desenvolvedor é ajudar a sua empresa a ser bem-sucedida. Segundo seu artigo, "não existe decisão técnica". Se, tecnicamente, X é mais aconselhável, mas Y é melhor para o negócio, então a decisão a ser tomada, fatalmente, **deve ser Y**.

Figura 21.3: Sistemas reativos, não confundir com sistemas remunerativos.

O negócio é o propósito do software, a razão de ser. Esse é precisamente nosso papel: alinhar as necessidades do negócio com a tecnologia, e não o oposto.

21.4 Conclusão

Como dito no primeiríssimo capítulo do livro, não dá para avaliar a qualidade sem saber qual a finalidade. É com esse propósito que analisamos a qualidade de um sistema reativo, a responsividade.

Requisições por segundo, Java, Node, Go? Spring Boot, PostgreSQL, MongoDB, AWS, GCP Azure, elasticidade e resiliência? Esquece! Nos sistemas reativos, nada disso é tão relevante quanto a responsividade para a satisfação do usuário.

Como vimos no capítulo *A Reatividade diz "olar"*, o Manifesto Reativo, literalmente, diz que, para sermos responsivos, precisamos das mensagens assíncronas, ser resilientes e elásticos.

Dizer que o sistema é responsivo significa que os seus desenvolvedores foram capazes de resolver muitos dos desafios que a capacidade de resposta traz: alta demanda do usuário, tratamento de falhas, consistência, distribuição, capacidade de lidar com cargas extremas etc.

Tudo isso faz parte da capacidade de resposta, da responsividade e aumenta a satisfação do usuário. Portanto, as demais propriedades reativas são o meio pelo qual atingiremos a responsividade e não o fim. Logo, elas têm sua importância, mas, definitivamente, não são a prioridade.

Figura 21.4: Elasticidade e resiliência não importam? Hã?

Eu sei que parece contraditório. Principalmente porque este parágrafo se encontra quase no final do livro quando todos os outros pontos do manifesto foram exaustivamente abordados. Mas, enquanto desenvolvedores, não somos pagos para desenhar gráficos reluzentes ou elaborar sistemas robustos. Somos pagos para resolver problemas.

Os sistemas são produtos do desperdício de descobrir uma solução. Se você pode resolver um problema com eficácia sem escrever um único sistema, ótimo! Entretanto, alguns problemas requerem sistemas para serem resolvidos e nesse momento, como gosta de dizer um amigo meu: damos o nosso show.

Parte 8: Considerações finais

Capítulo 22
Fim

Uma vez que o assunto é vasto e conectado a muitos outros, a tarefa de abordá-los por completo é ingrata, difícil e, principalmente, maior do que caberia neste livro.

Por maior que seja minha boa vontade, que não é tão grande assim, alguns assuntos relacionados foram mencionados apenas superficialmente. Destaco *event-sourcing*, streams reativos, modelo de atores, sistemas baseados em fila, arquitetura evolutiva e outros tópicos como material que possuem relação com o tema e que podem interessar aos leitores e leitoras.

Se eu os abordasse mais profundamente, este livro seria muito grande e não cumpriria um dos objetivos mais nobres de todos os livros: ser lido até o final. Além disso, eu preciso de um bom *cliffhanger* para deixar vocês ansiosos por um eventual livro novo... Deixo essa informação aí no ar.

Conclusão

Aos leitores e leitoras que resistiram às inúmeras piadas de mau gosto inseridas no livro, meus sinceros parabéns e minhas desculpas.

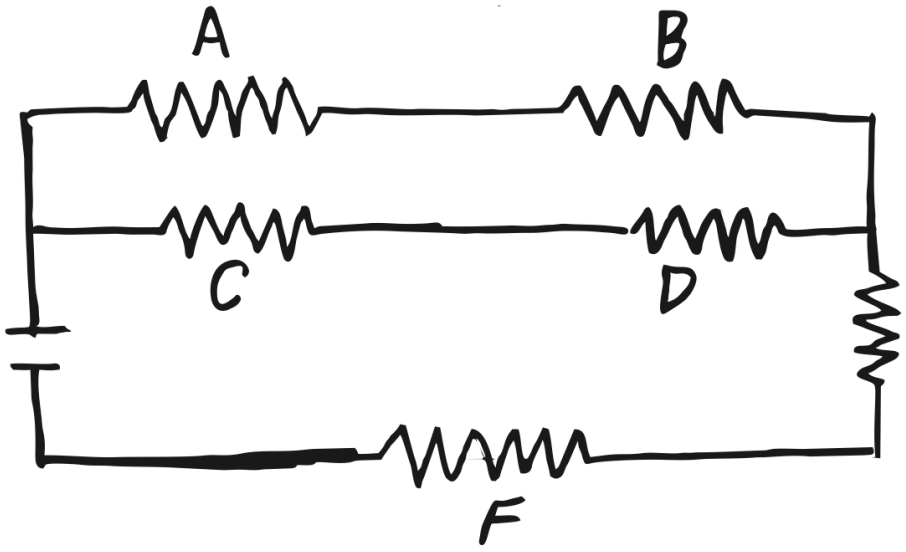

Figura 22.1: Sistemas reativos, não confundir com sistemas resistivos.

Espero que tenham considerado a leitura valiosa, apesar dos pesares, e tenham tirado ensinamentos proveitosos nas horas despendidas ao longo dessas páginas.

Vocês podem me encontrar facilmente em https://pagehub.me/virgs ou me contatar por email em guilherme.moraes@outlook.com, qualquer feedback é bem-vindo.

Falou, valeu e bebam água!

Capítulo 23
Bibliografia

ALLEN, Jamie; HANAFEE, Brian; KUHN, Roland. *Reactive design patterns*. 1 ed. Shelter Island: Manning Publications, 2016.

AMDAHL, Gene M., *The Logical Design of an Intermediate Speed Digital Computer*. **University of Wisconsin**. Outubro de 1951. http://pages.cs.wisc.edu/~bezenek/Stuff/amdahl_thesis.pdf. Acessado em 19 de abril de 2021.

BEYER, Betsy. et al. *Engenharia de confiabilidade do google*: como o Google administra seus sistemas de produção. 1 ed. São Paulo: Novatec Editora, 2016.

BONÉR, Jonas. *Reactive microsystems*: the evolution of microservices at scale. 1 ed. Sebastopol: O'Reilly Media, 2017.

CAMPBELL, Dave; HELLAND, Pat. *Building on Quicksand*. 2009.https://database.cs.wisc.edu/cidr/cidr2009/Paper_133.pdf. Acessado em 19 de abril de 2021.

FOWLER, Martin. *Circuit breaker*. **MartinFowler**, 2014.https://martinfowler.com/bliki/CircuitBreaker.html. Acessado em 19 de abril de 2021.

FOWLER, Martin. *Command query separation*. **MartinFowler**, 2005.https://martinfowler.com/bliki/CommandQuerySeparation.html. Acessado em 19 de abril de 2021.

FOWLER, Martin. *First law*. **MartinFowler**. https://martinfowler.com/bliki/FirstLaw.html. Acessado em 19 de abril de 2021.

FOWLER, Martin. *Getter eradicator*. **MartinFowler**, 2006 https://martinfowler.com/bliki/GetterEradicator.html. Acessado em 24

de maio de 2021.

FOWLER, Martin; LEWIS, James. *Microservices: a definition of this new architectural term*. **MartinFowler**, 2014. https://martinfowler.com/articles/microservices.html. Acessado em 19 de abril de 2021.

FOWLER, Martin. *Refatoração*: Aperfeiçoando o design de códigos existentes. 2 ed. São Paulo: Novatec Editora, 2020.

FOWLER, Martin. *Two hard things*. **MartinFowler**, 2009. https://martinfowler.com/bliki/TwoHardThings.html. Acessado em 19 de abril de 2021.

FRIEDRICHSEN, Uwe. *Towards successful resilient software design*. **InfoQ**, 2019. https://www.infoq.com/articles/towards-resilient-software-design. Acessado em 19 de abril de 2021.

FROES, Guilherme. *There's no such thing as a technical decision*. **gui froes**, 2019. https://guifroes.com/no-such-thing. Acessado em 19 de abril de 2021.

GAMMA, Erich et. al. *Padrões de projeto*: soluções reutilizáveis de software orientado a objetos. 1 ed. Porto Alegre: Grupo A, 2000.

GARCIA-MOLINA, Hector; SALEM, Kenneth. *SAGAS*. *Department of Computer Science*. *Princeton University*, 1987. http://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf. Acessado em 19 de abril de 2021.

GILBERT, Seth; LYNCH, Nancy. *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*. ACM SIGACT News, 2002.

GREGG, Brendan. *The USE method*. **brendangregg**, 2017. http://www.brendangregg.com/usemethod.html. Acessado em 19 de abril de 2021.

HANMER, Robert S. *Patterns for fault tolerant software*. 1 ed. Hoboken: Wiley, 2007.

HOHPE, Gregor; WOOLF, Bobby. *Enterprise integration patterns*: designing, building, and deploying messaging solutions. 1 ed. Boston: Addison-Wesley Professional, 2003.

KLEPPMANN, Martin. *Designing data-intensive applications*: the big ideas behind reliable, scalable, and maintainable systems_. 1 ed. Sebastopol: O'Reilly Media, 2017.

KRZYZANOWSKI, Brendan. *Distributed transactions: distributed deadlock*. **Paul Krzyzanowski**, 2017. https://www.cs.rutgers.edu/~pxk/417/notes/deadlock.html. Acessado em 19 de abril de 2021.

MARTIN, Robert C. *Arquitetura limpa*: o Guia do Artesão Para Estrutura e Design de Software. 1 ed. Rio de Janeiro: Alta Books, 2019.

MARTIN, Robert C. *Código limpo*: habilidades práticas do Agile Software. 1 ed. Rio de Janeiro: Alta Books, 2009.

NEWMAN, Sam. *Building microservices*: designing fine-grained systems. 1 ed. Sebastopol: O'Reilly Media, 2015.

NEWMAN, Sam. *Monolith to microservices*: evolutionary patterns to transform your monolith. 1 ed. Sebastopol: O'Reilly Media, 2019.

NYGARD, Michael T. *Release It!*: Design and deploy production-ready software. 1 ed. Raleigh: Pragmatic Bookshelf, 2007.

NYSTROM, Robert. *Game programming patterns*. 1 ed. Genever Benning, 2014.

PARSONS, Rebecca. *Adotar microsserviços?*. **ThoughtWorks**, 2018. https://www.thoughtworks.com/pt/insights/blog/microservices-adopt. Acessado em 19 de abril de 2021.

RODRIGUES, Caique. *A arte da programação*. **Caiquerodrigues**, 2019. https://caiquerodrigues.me/a-arte-da-programacao. Acessado em 19 de abril de 2021.

SRIDHARAN, Cindy. *Distributed systems observability*. 1 ed. Sebastopol: O'Reilly Media, 2018.

STOPFORD, Ben. *Designing event-driven systems*: concepts and patterns for streaming services with apache kafka. 1 ed. Sebastopol: O'Reilly Media, 2018.

THOMPSON, Martin. *Single Writer Principle*. **mechanical-sympathy**, 2011.

https://mechanical-sympathy.blogspot.com/2011/09/single-writer-principle.html. Acessado em 19 de abril de 2021.

VITILLO, Roberto. *Understanding distributed systems*: what every developer should know about large distributed applications_. 1 ed. Roberto Vitillo, 2021.

WILKIE, Tom. *The RED Method: key metrics for microservices architecture*. **Weave Works**, 2017. https://www.weave.works/blog/the-red-method-key-metrics-for-microservices-architecture. Acessado em 19 de abril de 2021.